ACM *OOPS Messenger* **4**,4 (Oct. 1993), 2-27.

Equal Rights for Functional Objects[1] or, The More Things Change, The More They Are the Same[2]

Henry G. Baker

Nimble Computer Corporation
16231 Meadow Ridge Way, Encino, CA 91436 (818) 986-1436 (818) 986-1360 FAX

August, October, 1990, October, 1991, and April, 1992

This work was supported in part by the U.S. Department of Energy Contract No. DE-AC03-88ER80663

We argue that intensional *object identity* in object-oriented programming languages and databases is best defined operationally by side-effect semantics. A corollary is that "functional" objects have extensional semantics. This model of object identity, which is analogous to the normal forms of relational algebra, provides cleaner semantics for the value-transmission operations and built-in primitive equality predicate of a programming language, and eliminates the confusion surrounding "call-by-value" and "call-by-reference" as well as the confusion of multiple equality predicates.

Implementation issues are discussed, and this model is shown to have significant performance advantages in persistent, parallel, distributed and multilingual processing environments. This model also provides insight into the "type equivalence" problem of Algol-68, Pascal and Ada.

1. INTRODUCTION

Parallel, distributed and persistent programming languages are leaving the laboratories for more wide-spread use. Due to the substantial differences between the semantics and implementations of these languages and traditional serial programming languages, however, some of the most basic notions of programming languages must be refined to allow efficient, *portable* implementations. In this paper, we are concerned with defining *object identity* in parallel, distributed and persistent systems in such a way that the intuitive semantics of serial implementations are transparently preserved. Great hardware effort and expense—e.g., cache coherency protocols for shared memory multiprocessors—are the result of this desire for transparency. Yet much of the synchronization cost of these protocols is wasted on *functional/immutable* objects, which do not have a coherency problem. If programming languages distinguish functional/immutable objects from non-functional/mutable objects, and if programs utilize a "mostly functional" style, then such programs will be efficient even in a non-shared-memory ("message-passing") implementation. Since it is likely that a cache-coherent shared-memory paradigm will not apply to a large fraction of distributed and persistent applications, our treatment of object identity provides increased cleanliness and efficiency even for non-shared-memory applications.

The most intuitive notion of object identity is offered by simple Smalltalk implementations in which "everything is a pointer". In these systems, an "object" is a sequence of locations in memory, and all "values" are homogeneously implemented as addresses (pointers) of such "objects". There are several serious problems with this model. First, "objects" in two different locations may have the same bit pattern both representing the integer "9"; an implementation must either make sure that copies like this cannot happen, or fix the equality comparison to dereference the pointers in this case. Second, the "everything is a pointer" model often entails an "everything is heap-allocated" policy, with its attendant overheads; an efficient implementation might wish to manage small fixed-size "things" like complex floating point numbers directly, rather than through pointers. Third, read access to the bits of an object may become a bottleneck in a multiprocessor environment due to locking and memory contention, even when the object is functional/immutable and could be transparently copied. In light of these problems, we seek a more efficient and less implementation-dependent notion of object identity than that of an address in a random-access computer memory.

A more efficient, but also more confusing, notion of object identity is offered by languages such as Pascal, Ada and C. These languages can be more efficient because they directly manipulate values other than pointers. This efficiency is gained, however, at the cost of an implementation-dependent notion of object identity. To a first

[1]"Functional objects" is triply overloaded, meaning immutable objects, function closures or objects with functional dependencies.

[2]*Plus ça change, plus c'est la même chose*—Alphonse Karr, as translated in [Cohen60,p.214].

approximation, a *value* in these languages is a *fixed-length* configuration of bits which can be manipulated directly, while an *object* is a configuration of bits which is manipulated through a pointer. The restriction on *values* to have only fixed lengths known to the compiler eliminates the possibility of variable-length functional/immutable *values*—e.g., character strings whose length cannot be determined at compile time; such values must be first-class (side-effectable) "objects" manipulated through pointers. The storage allocation and potential dangling reference problems of pointers cause these languages great uneasiness, however. Ada's paranoia about pointers leads to its bizarre and error-prone "**in out**" parameter passing, which it uses in preference to "by-reference" parameter passing, but whose semantics do not preserve object identity. The copying implicit in Ada's "**in out**" parameter passing can be characterized as a clumsy attempt at a software "cache coherency protocol", but it fails to provide the desired transparency.

We call the problem of providing clean, efficient semantics for "object identity" the *object identity crisis*, because the costs of full object identity in persistent, parallel and distributed systems are far greater than in serial, single-process systems. We will argue that the notions of "object identity" and "distinguishable by side-effects/assignment" are equivalent, and that applying this equivalence provides cleaner semantics for argument-passing, result-returning, and the built-in "equality" predicate of a programming language. Our model solves the problem of integrating functional (immutable) objects (e.g., numbers, strings) with non-functional objects by providing all objects with "object identity", but without the usual costs of full object status. This model of object identity also offers an interesting insight into the problem of "type equivalence".

Most of our examples and discussions will center on Common Lisp, because Common Lisp exhibits the wide range of issues we are trying to resolve, Lisp has a well-known, trivial syntax, Lisp has a universal type, and Lisp is intuitively defined in Lisp itself in a "meta-circular" manner. These properties allow the issues to be discussed with a minimum of extraneous detail. Our model is easily transported, however, to any other language in which object identity must be defined.

2. OBJECT IDENTITY

Modern "object-oriented" programming languages and data bases are based upon the notion of "object identity" [Khoshafian86] [King89] [Ohori90]. Every "object" has an identity that is unique even when its attributes are not. For example, object A and object B are distinguishable even when they have the same attributes. In a sense, object identity can be considered to be a rejection of the "relational algebra" view of the world [Ullman80] in which two objects can only be distinguished through differing attributes.[3]

Distinguishing objects can be done in several ways, with the "true" notion of object identity achieved through the finest distinction. Many programming languages provide a primitive equality predicate which can be used to distinguish objects. For example, most "systems programming" languages such as C or Ada compare simple objects like fixed-precision integers and characters for bit-representation equality, and pointers for location equality. As we will show, however, these predicates are often unreliable, in the sense that they sometimes make distinctions that are not otherwise visible, they sometimes fail to distinguish distinct objects, and they sometimes just plain fail. A more reliable, but more expensive, test for identity is "operational equivalence" [Rees86], invented by Morris [Morris68] and popularized by Plotkin [Plotkin75]; the Scheme Report defines operational equivalence as follows:

> Two objects are "operationally equivalent" if and only if there is no way that they can be distinguished, using ... primitives other than [equality primitives]. It is guaranteed that objects maintain their operational identity despite being named by variables or fetched from or stored into data structures. [Rees86,6.2]

This description of "operational identity" given above incorporates some of the most basic notions of "object identity":

- objects retain their identity throughout their lifetime—identity is constant and immutable;
- object identity is preserved by the basic value-transmission operations of the programming language—binding, argument-passing, value-returning, and assignment;[4]
- objects with the same identity are operationally equivalent.

---

[3]Object identity can be grafted onto a relational system through the use of a "time stamp" or equivalent attribute. This "attribute" is variously called an *oid*, a *surrogate*, an *l-value*, or an *object identifier* [Abiteboul89].

[4]Any value-transmission operation that does not preserve object identity is therefore an implicit *coercion*. Since implicit coercions are some of the more error-prone features of programming languages [DoD78,3B] [Radin78], we propose that coercions in the basic value transmitting operations of programming languages be eliminated.

We would like any predicate `e` for testing object identity to meet the following requirements, which provide bounds on the notion of object identity:

- `e` is an *equivalence relation*: e(x,x); e(x,y)⇒e(y,x); and e(x,y)∧e(y,z)⇒e(x,z)
- e(x,y) iff (let z=x in e(z,y)) iff (λ(z) e(z,y))(x) iff e((λ() x)(),y) iff (let z in z:=x; e(z,y))
- if (λ(x) f(x)) is a constant (single-valued) *function*, and if e(x,y), then e(f(x),f(y))
- if `c` is a *constructor* of immutable objects, then `c` is a constant (mathematical) function.[5]

Our requirements for an object identity predicate allow for an equivalence relation that is finer than operational equivalence. Since operational equivalence is undecidable, any decidable test for object identity will have to be strictly finer than operational equivalence. As a result, some functional objects, for example, will be considered distinct even though any application of these objects to identical arguments will produce identical results. However, object identity, as defined by an equality predicate, will be constant, and referential transparency for mathematical functions will be observed.

Lisp is the first major computer language to provide "object identity". A Lisp list can have the same elements in the same order, but it is not necessarily the "same" list. The built-in Lisp predicate `EQ` can be used to make this distinction, but the distinction can be more reliably drawn in an operational fashion using update semantics for determining object identity. Consider the following operational definition of `EQ` for cons cells:[6]

```
(defun eq-cons (x y)
  (let ((oldcar (car x)))
    (rplaca x 'private-symbol-for-eq-cons)
    (prog1 (eq-symbol (car y) 'private-symbol-for-eq-cons)
           (rplaca x oldcar))))
```

The experienced Lisp programmer may not have seen this particular pedagogical definition of `EQ`, but side-effecting functions like `NREVERSE` cannot be properly understood without this intuition.

A considerable amount of effort in beginning Lisp courses is devoted to providing the student a proper model for cons cells [Abelson85,3.3]. The two popular models are the "box-and-pointer diagrams" model introduced in [McCarthy60] and the linear-array-of-pointer-pairs model. Both of these models provide the correct operational semantics for `EQ`, because the `CAR` of a cell can be affected if and only if one applies `RPLACA` to "the" cons cell. Once these models are understood, `EQ` is a well-defined and natural predicate on mutable structured objects.[7]

The Lisp predicate `EQUAL` greatly confuses the situation, however. `EQUAL` tests for structural similarity rather than object identity, by recursing on the `CAR`'s and `CDR`'s of the sub-expressions:

```
(defun equal (x y)
  (or (and (atom x) (atom y) (eq-atom x y))
      (and (equal (car x) (car y))
           (equal (cdr x) (cdr y)))))
```

There are a several problems with `EQUAL`. First, it may diverge in the presence of directed cycles (loops) in one of its arguments, although some (e.g., [Pacini78]) have suggested more sophisticated predicates capable of detecting such cycles. Secondly, it is not *referentially transparent*; two calls to `EQUAL` on the "same" (i.e., `EQ`) arguments can produce different results at different times. Neither of these possibilities is to be desired in a programming language primitive equality test because we would like such a test to always return and we would like object identity to be preserved.

5[Ohori89] states that different invocations of *mutable* constructors are guaranteed to produce distinct results.

6Mason's *defined:eq* [Mason86,p.66] uses the same technique. McCarthy's original "recursive functions" paper [McCarthy60] explicitly makes `EQ` *undefined* for cons cells, where the term "undefined" to the recursion theorist typically means "diverges" or "loops". McCarthy's paper also describes a *garbage collection* algorithm which twiddles an object's mark bit, thereby demonstrating the use of side-effects to define object identity.

7Garbage collectors have never trusted equality predicates to determine object identity, but have always used side-effect semantics; that's what "mark bits" and "reference counts" are for! A garbage collector can be built which utilizes an equality predicate instead of a mark bit—e.g., by using a hash table instead of a mark bit—but such a garbage collector would likely be slow. We argue later that a garbage collector can sometimes achieve increased performance by not preserving an identity predicate finer than the `egal` predicate defined later.

Yet EQUAL is an extremely valuable operation, because the vast majority of Lisp lists are side-effect free—i.e., *"pure"*. Without side-effects, loops cannot be constructed and sublists cannot be modified, and within these restrictions EQUAL becomes a well-defined and useful operation.[8]

This tension between the simplicity of EQ and the usefulness of EQUAL has led to a great deal of confusion. This confusion has now lasted for 30 years, with additional confusion having been recently added by Common Lisp. Since neither EQ nor EQUAL has the desired semantics for the multiplicity of Common Lisp datatypes, Common Lisp added six more—EQL, EQUALP, =, CHAR=, STRING=, and STRING-EQUAL, yet these also lack the correct semantics.[9] Scheme continues this confusion by offering the three predicates (eq?, eqv? and equal?) which are roughly equivalent to Common Lisp's EQ, EQL and EQUALP.

Why have so many different notions of equality been defined? We believe that this confusion has been caused by the tension between clean semantics and efficient execution. While every Lisp object can be provided with "object identity" by allocating a separate address for each number and character, such a Lisp is extremely slow because every arithmetic operation is accompanied by a memory allocation operation that is required to hold the new number.[10] However, the overhead of providing every number with "object identity" seems silly when the typical number is itself smaller than an object pointer. This space overhead, combined with the time overhead for allocation and dereferencing, has led to modern Lisps wherein the bits for a number's representation are stored in pointer-like form, with no allocation or dereferencing necessary—e.g., Interlisp's *inums* [Teitelman74]. When represented in this "immediate" format, numbers can be compared with EQ, which typically performs a simple pointer comparison, instead of EQUAL, which would require dereferencing.

Unfortunately, not all numbers can be represented in such a compact format; e.g., "bignums"—extended precision integers—require the allocation of additional storage, and hence require the use of a true pointer in order to avoid manipulating variable-length objects. Rather than requiring a programmer to constantly test whether an integer is large or small before making the appropriate comparison, Common Lisp defines the EQL predicate, which dereferences numbers when necessary.[11] Dereferencing when comparing numbers is consistent, however, because numbers are *immutable* objects whose components cannot be side-effected.[12] Thus, two instances of a number can sometimes be distinguished using EQ when they are supposed to be the same number; EQ draws too fine a distinction in this case.[13]

Strings provide more examples of EQ/EQUAL confusion. We would like to provide a primitive mechanism for comparing strings which treats each string as a whole, without requiring us to iterate through the individual string elements. In addition to the general usefulness of such a predicate, character strings and bit strings can often be compared more efficiently by considering the characters in groups rather than individually; this desire for efficiency usually results in this predicate becoming a primitive [Baker90b]. Common Lisp strings are instances of Common Lisp arrays, however, and hence their elements can be changed through assignment. By our operational definition of EQ given above, therefore, different strings are different, even when their spellings are currently the same.

Most strings, however, are *constant*—i.e., their spellings are never modified. The Common Lisp semantics which provide first-class object identity for strings means that storage allocation and dereferencing is always required, even when the strings are small and constant. In a distributed programming environment in which strings are incorporated

---

[8]McCarthy's EQUAL function [McCarthy60] applies only to immutable cons cells, and within that context, his EQUAL meets our requirements. He purposely crippled EQ (hence its name) to apply only to non-cons cells. EQ does not meet our requirements for object identity for pure cons cells because in most Lisp systems (EQ (CONS x y) (CONS x y)) yields false—i.e., CONS is not a function.

[9]Steele reports that the ANSI Common Lisp committee concluded that "object equality is not a concept for which there is a uniquely determined correct algorithm" [Steele90,p.109]. This paper suggests otherwise.

[10]*MacLisp* [Moon74] numbers were allocated in this fashion, but small integers were "uniquized" with a table.

[11]Dereferencing can be avoided, allowing EQ to be universally used for comparing numbers, if all numbers are *uniquized*. Uniquization is the same process whereby symbols which are spelled the same are required to point to the same address, and hence be the same object. However, uniquization for arbitrary numbers has traditionally been found to be more expensive than dereferencing.

[12]Interlisp's setn primitive allows for unspeakable violence to numbers; its use is reserved for those who mutilate Fortran's call-by-reference constants by the light of the full moon.

[13]This superfine distinction can be embarrassing, as Common Lisp is allowed to copy numbers whenever it feels the urge [Steele90,p.104]. The noise often heard when this kind of bug is discovered is "eek!".

into messages, the overhead of providing EQ-style object identity for strings can be extremely expensive. For this reason, Cedar [Swinehart86] offers the notion of "ropes", which are functional (immutable) strings, and hence can be copied at will.[14] Unfortunately, neither Common Lisp nor Scheme has yet seen fit to provide for functional strings, although *AutoLISP* [Autodesk88] has *only* functional strings. Thus, strings are another datatype in which EQ is often too fine, and EQUAL is often too coarse.

The representation of numbers and strings are instances of a more generic functional representation problem. Whether functional objects are to be represented directly or as pointers is an implementation issue that should be hidden from the programmer. Therefore, such notions as "shallow equality" and "deep equality" [Atkinson89] are wrong-headed, because they allow implementation decisions to "leak through" to the programmer. For example, an object with a functional attribute might represent the attribute directly if it is small and if it belongs to a type which can be represented in a number of bits which can be fixed at compile time; alternatively, it could represent the attribute through a pointer to another functional object. Using shallow and deep equality predicates, however, the programmer could distinguish these representations, even though this distinction can only confuse him since the objects are functional.

Common Lisp *hash tables* and *property lists* present another important identity problem; an important use of hash tables is to give properties to non-symbols. The choice of the proper predicate to use (EQ, EQL, EQUAL, etc.) for the hash table/property list depends upon the kinds of objects to be used as keys: numbers should use EQL, while strings should use EQUAL. Hashing a number in an EQ-hash table will probably result in losing the property, because numbers which are EQL are not necessarily EQ; whether two numbers remain EQ over time is not guaranteed. Hashing a string into an EQUAL-hash table does not actually hash the string itself, but the functional *contents* of the string. If the implementation erroneously hashed the string itself, then a later modification of the string would cause the hash table code to crash or otherwise fail. Unfortunately, Common Lisp does not offer a single kind of hash table which will work uniformly and reliably on every object; such a hash table would be required for a generic "memoizing" function [Bird80]. We later define a predicate EGAL for the universal determination of object identity which can be used for such generic memoizing.

In the next section, we argue that neither EQ nor EQUAL is wrong; EQ is correct for mutable cons cells and EQUAL is correct for immutable cons cells. The major mistake of Lisp is in not distinguishing the two kinds of cons cells based on their mutability. Goto [Goto74] [Goto76] introduced this distinction, but for a different, although related, purpose.

3. OUR MODEL OF OBJECT IDENTITY

A. Mutability Definition of Object Identity

Our model for object identity is similar to Scheme's concept of "operational identity" [Rees86], in which objects which *behave* the same should *be* the same. However, since "behave the same" is undecidable for functions and function-closures, we back down from "operational identity" to "operational identity of data structure representations". Operational identity for data structures is much easier than operational identity for function-closures, because there are only a few well-defined operations on data structures, but function-closures can do anything. We define a single, computable, primitive equality predicate called EGAL which we show is consistent with the notion of "operational identity of data structures". *Egal* is the obsolete Norman term for *equal*, and *Égalité* is the French word for social equality. "During the seventeenth century *two parallel vertical lines* were frequently used [to denote equality], especially in France, instead of =" [Young11]; we will later find that ‖ is a remarkably satisfying infix symbol for *egal*.

Our model for object identity distinguishes mutable objects from immutable objects, and mutable components of aggregate objects from immutable components. We consider an immutable component of an object to be an integral part of the object's identity, since it cannot be separated from the object. Unlike a normalized (factored) relational database, which attempts to *minimize* the size of a "key" which holds the essence of an entity [Ullman80,s.5.4], we *maximize* the size of the object "key" to include all of its static components. Because these components are static, we cannot create any "update anomalies" with this policy. In particular, this object identity can be used as a key to a Common Lisp hash table [Steele90,p.435], and no hash entries will become inaccessible as a result of a key element being modified.

[14]The term "rope" is curious; presumably functional ropes are "lighter-weight" than mutable strings. "Thread" would have been a better choice, but that term was already taken. "Filament" might have been the best choice. Common Lisp *keywords* are rough analogues to Cedar ropes.

We will build up our definition of `EGAL` incrementally, starting from a relatively simple base. An object is *immutable* if all of its (top-level) components are immutable, otherwise it is *mutable*. Briefly, if an object is mutable, then we compare it using an "address-like" comparison, while if an object is immutable, then we recursively compare its components. This recursion is only used to *define* the semantics of `EGAL`; a given implementation may not need to recurse. For example, Goto's "hash consing" [Goto74] [Goto76] allows functional lists to be compared without recursion. Below is a first approximation to `EGAL`:

```
(defun egal (x y)
  (and (egal (type-of x) (type-of y))
       (cond ((symbolp x)       (eq x y))
             ((numberp x)       (egal-number x y))
             ((consp x)         (eq x y))
             ((vectorp x)       (egal-vector x y))
             ((functionp x)     (egal-function x y))
             ((hash-table-p x)  (egal-hashtable x y))
             ((streamp x)       (egal-stream x y))
             ((mutable-structure-p x)
                                (eq x y))
             (t                 (every #'(lambda (component)
                                           (egal (funcall component x)
                                                 (funcall component y)))
                                       (components (type-of x)))))))
```

B. Equality of Symbols and Numbers

The first clause of `EGAL` concerning symbols is not strictly necessary, since symbols are mutable (i.e., their values and properties can be changed), and hence they are included in the set of mutable structures. The second clause concerning numbers is also not strictly necessary, because numbers are immutable and are handled by the last clause. Since there are many types of Common Lisp numbers, however, we explicitly show how to compare them, as follows:

```
(defun egal-number (x y)
  (and (egal (type-of x) (type-of y))
       (cond ((complexp x)
              (and (egal-number (realpart x) (realpart y))
                   (egal-number (imagpart x) (imagpart y))))
             ((rationalp x)
              (and (egal-number (numerator x) (numerator y))
                   (egal-number (denominator x) (denominator y))))
             ((floatp x)
              (and (= (float-sign x) (float-sign y)) ; for IEEE-754[15]
                   (= x y)))
             ((and (fixnump x) (fixnump y)) (eq x y))
             ((and (bignump x) (bignump y))
              (every #'eq (digits x) (digits y)))
             (t nil))))
```

The use of strict equality "=" to compare two floating-point numbers is essential for object identity. In particular, the unfortunate tendency of some programming languages (e.g., APL) to include some sort of "fuzz" in the comparison of floating-point numbers is not acceptable. This is because such a fuzzy comparison is an *analytic* notion of closeness, and not an *algebraic* equivalence relation. In particular, fuzzy comparisons are not transitive. Since one of our goals is to preserve the semantics of table lookup, a fuzzy comparison would make such a notion ill-defined, because there may be several table keys which are sufficiently close to a test key. If a programmer or language designer wishes to have a fuzzy comparison, it should be a predicate distinct from the object identity predicate.

---

[15] Either IEEE = is not an identity predicate or IEEE `atan` is not a function, because `0.0=-0.0`, but `atan(0.0,-0.0)≠atan(-0.0,-0.0)`; due to this and other reasons, IEEE `-0.0` is an *algebraic abomination*.

Common Lisp does not support user-defined objects analogous to bignums and complex numbers, which would be `EQL` but not necessarily `EQ`; `EQUALP` descends into structures regardless of mutability. Since none of these equality predicates are CLOS generic functions [Steele90,ch.28], their behavior cannot be overloaded by user-supplied methods. The above definition for `EGAL`, however, allows the user to trivially define functional rational and complex number types himself using `defstruct` as follows, because `EGAL` will automatically recurse into the components of an immutable object.

```
(defstruct (complex
             (:constructor complex (realpart &optional (imagpart 0))))
  "Immutable (functional) complex numbers."
  (realpart 0 :read-only t :type real)
  (imagpart 0 :read-only t :type real))

(defstruct (ratio
             (:constructor / (numerator denominator)))
  "Immutable (functional) ratio numbers."
  (numerator   0 :read-only t :type integer)
  (denominator 1 :read-only t :type integer))
```

C. Equality of Vectors and Strings

If Common Lisp defined immutable vectors (and hence immutable strings), `EGAL` would utilize the following code:

```
(defun egal-vector (x y)
  (cond ((and (mutable-vector-p x) (mutable-vector-p y))
         (eq x y))
        ((and (immutable-vector-p x) (immutable-vector-p y))
         (and (= (length x) (length y))
              (dotimes (i (length x) t)
                (unless (egal (aref x i) (aref y i))
                  (return nil)))))
        (t nil)))
```

D. Equality of Functions and Function-Closures

The most problematic datatypes for equality are the function objects—*simple* functions ("compiled-functions") and function *closures* ("FUNARGS" [Moses70]). We would like to compare closures properly because they can be used for object-oriented programming based on *delegation* instead of *inheritance* [Snyder86] [Lieberman86] [Ungar87]. *Simple* functions access only their arguments and global variables, and therefore require no environment pointer; e.g., the C language [ANSI-C] provides simple function objects, but not function closures. Simple function objects in a functional language which do not reference global names are called *combinators*. Function closures are data structures incorporating both a simple function and an "environment" mechanism which provides the values of the "free variables" in the simple function; e.g., Algol-68, Pascal and Lisp offer lexical closures; Ada can approximate function closures using *tasks* [Lamb83].

True operational equivalence for function objects—even for simple functions—is impossible to compute, because the equivalence of programs is undecidable. Rather than "throw out the baby with the bath water" by refusing to compare functions, however, we desire to compare function closures in the same manner as the data structures that they may emulate. Consider the following ("object-oriented") simulation of Lisp's cons cells:

```
(defun cons (x y)
  #'(lambda (m &optional z)
      (caseq m (car x)
               (cdr y)
               (consp t)
               (rplaca (setq x z) nil)
               (rplacd (setq y z) nil))))

(defun car (c) (funcall c 'car))

(defun cdr (c) (funcall c 'cdr))

(defun rplaca (c x) (funcall c 'rplaca x))

(defun rplacd (c y) (funcall c 'rplacd y))
```


Since these simulated cons cells are mutable, we would like each of them to have a separate object identity, since they can be distinguised through mutations. Since "`#'`" ("`FUNCTION`") constructs a new function-closure object, Common Lisp's `EQ` has the correct semantics for this case. Consider, however, the simulation of *immutable* list cells:

```
(defun functional-cons (x y)
  #'(lambda (m)
      (caseq m (car x)
               (cdr y)
               (consp t))))

(defun car (c) (funcall c 'car))

(defun cdr (c) (funcall c 'cdr))
```

These cons cells are "read-only", since there are no `setq`'s to the free variables in the closure. We desire the `EGAL` predicate to perform correctly in this case, as well as in the mutable case above. `EGAL` can only get the correct answer if closures themselves are "functional" objects which allow `EGAL` to recursively examine their structure. In the case of our functional cons, the closure consists of 3 elements: the code for the closure and the values of `x` and `y`. If the components `x` and `y` are immutable, then the closure itself is immutable, because the code pointer cannot be changed. We therefore define `EGAL` on functions and function-closures as follows:

```
(defun egal-function (x y)
  (and (egal (type-of x) (type-of y))
       ((simple-function-p x) (eq x y))
       ((closure-p x)
        (and (egal-function (code x) (code y))
             (egal-environment (env x) (env y))))))
```

This definition for `EGAL` will work properly on both versions of `cons` because while the closure itself is in both cases immutable, the environment in the mutable case will itself be mutable, while the environment in the immutable case will be immutable. We have only succeeded in transforming the equality problem of closures into the equality problem of environments, however.

In order to understand how to compare environments, we must first look at their structure. An environment defines the values of the variables which are "free" in a "code" object. This definition can be in the form of an "association list", which is a list of pairs <variable-name,value>, which is searched to find the value, or the environment can be in the form of a vector which is indexed by an index number associated with each variable. Unfortunately, some language interpreters include in a closure environment values for variables which are not needed by the closure code. While these inclusions may cause excessive storage usage [Baker92a], they should not affect the course of the computation. We must therefore make sure that these extraneous values in closure environments do not affect the result of our equality comparison.

Extraneous values can be ignored in two ways: by making sure that the closures are not constructed with extraneous values, or by modifying environment comparison to ignore these extraneous values. Many Common Lisp compilers implement closures without extraneous values, but many Common Lisp interpreters implement closures with extraneous values.[16] Since we do not allow compiled closures to compare equal to interpreted closures, even when the compiled code resulted from compiling the interpreted code, we do not have to worry about comparing environments with incompatible structures. Furthermore, interpreted closures can compare equal only if they have *identical* (i.e., `EQ`) code—no matter how it looks when printed.[17] Therefore, the comparison of interpreted environments has to be performed in the presence of a list of "essential" (i.e., non-extraneous) values, as follows:

```
(defun egal-environment (vars x y)
  (every #'(lambda (v) (egal (lookup v x) (lookup v y)))
         vars))
```

---

[16]Some compilers perform the "optimization" of sharing environment structures among closures [Kranz86], even though this optimization may cause a form of "storage leak", wherein the extraneous objects avoid garbage collection [Baker92a].

[17]We later argue that Lisp source code should be functional; in this case, source code which prints the same will compare the same, i.e., be `EGAL`.

But wait! It would seem that this definition of environment comparison would allow `EGAL` to recurse on the comparison of our mutable cons cells as defined above. However, this is not correct. To achieve the correct semantics for mutable local variables which may be captured by closures, all Lisp systems perform a program transformation equivalent to that we call "cell introduction" [Sandewall74] [Kranz86]. Cell introduction provides for the binding of the assignable free variables to an anonymous "cell" which is then read and written instead of the variable itself. Below is the code for our mutable cell after cell introduction:

```
(defun cons (x y)
  (let ((x (make-cell x))
        (y (make-cell y)))
    #'(lambda (m &optional z)
        (caseq m (car (cell-value x))
                 (cdr (cell-value y))
                 (consp t)
                 (rplaca (setf (cell-value x) z) nil)
                 (rplacd (setf (cell-value y) z) nil)))))
```

As a result of cell introduction, the free variables `x,y` of our closure are now bound to cells instead of directly to their initial values. Thus, while `EGAL` on these environments will recurse down to the cell level, it will stop there because cells are mutable. Thus, even though the top-levels of closures created at different times will compare equal, the cells created by `make-cell` at these different times will not compare equal, so the semantics of `EGAL` are correct.

There remains a nagging problem with closure comparisons, however. The notion of a "free variable" of a lambda-expression is not well-defined, because a smart compiler may be able to eliminate all references to certain free variables. Such an optimization may change the behavior of `EGAL` in a way visible to the programmer if the closure becomes functional as a result of the optimization. One approach would be to include a variable in the closure if it appears free in the unoptimized source code, even if all of its occurences are later optimized away. Another approach would be to have the language provide a means for the programmer to declare which free variables he would like included in the closure, and thus participate in the `EGAL` comparison. An interpreter/compiler can easily check that this list is a superset of the variables which are free in the (optimized) code. Such a declaration was a necessity when Lisp variables were dynamically scoped, and is a welcome documentation aid for any lambda-expression in a lexically scoped Lisp. Below is shown one possible format for such a free variable declaration in Common Lisp:

```
#'(lambda (x)
    (declare (free a b c)) ; If a "free" declaration is given,
    (list x a b))          ; every free var must be in the list.
```

The final problem for the equivalence of functions involves *mutually recursive* functions. If the functions are known to the compiler to be simple, then the recursion may be closed by the assembler or linking loader, in which case `EGAL` will not be aware of the implicit recursive cycles. On the other hand, if a nest of mutually recursive functions are created by Common Lisp's `LABELS` (Scheme's `letrec`), then typical implementations will create cyclic environment structures to handle the recursion. Of course, cyclic environments are not required to implement recursive functions if we use the "Y combinator" trick from the lambda calculus [Gabriel88], as the factorial example below demonstrates.

```
(defun fact (f n) (if (= n 0) 1 (* n (funcall f (- n 1)))))

(defun factorial (n) (fact #'fact n))
```

Since our environment structures must be functional and finite, mutual recursion must therefore be implemented by means of the Y combinator shown for factorial, or by means of cyles created with side-effectable cells. Let us call the Y combinator method "pure" and the cyclic method "impure". Although the functions created using the two methods are "applicationally equivalent", their different natures can be distinguished by means of `EGAL`. Since the majority of language implementations utilize side-effects and cycles, and since it is difficult to hide the existence of the side-effectable cells used in these cyclic environments—e.g. Scheme's `letrec` [Bawden88], it is easier to use the impure method as the *definition* of mutual recursion. We therefore define Common Lisp's recursive `LABELS` in terms of non-recursive `FLET`, as follows:

```
(labels ((foo (x) << body of foo >>)
         (bar (y z) << body of bar >>))
  << body of labels using foo and bar. >>) ; is defined by:
```

```
(let ((foo-cell nil)
      (bar-cell nil))
  (flet ((foo (x) (funcall foo-cell x))
         (bar (y z) (funcall bar-cell y z)))
    (setq foo-cell #'(lambda (x) << body of foo >>)
          bar-cell #'(lambda (y z) << body of bar >>))
    << body of labels using foo and bar. >>))
```

A persistent, parallel and/or distributed object-oriented programming language might want to offer both "pure" and "impure" mechanisms for implementing mutual recursion, since their equivalence semantics would differ. The "impure" mechanism would offer traditional semantics, while the "pure" mechanism would exhibit functional behavior; this functional behavior being especially important in the case where the programmer was lazy and the functions did not actually call one another in a mutually recursive manner.

In summary, our model of equivalence for function closures is not based on "behavioral equivalence"—because that is undecidable—but on "representational equivalence". While "representational equivalence" provides a finer distinction than "operational equivalence", and may not provide all the equivalence we might like—e.g., equivalence under lexical variable "alpha-renaming"—we believe that "representational equivalence" provides an extremely useful extention of data structure equivalence into the realm of closures. Finally, "representational equivalence" does not prove embarrassing, since identity is constant and identical arguments to mathematical functions give identical results.

E. Equality of Hash Tables

Hash tables are a concrete implementation of an abstract function represented by the set of its <domain-element,range-element> association pairs. The hash tables of Common Lisp [Steele90,p.435] are mutable in two different ways. First, the domain of the function can be changed by adding or deleting association pairs. Second, the range-element associated with a particular domain-element can be changed. As a result of this modifiability, standard Common Lisp hash tables should compare `EGAL` if and only if they are identically the same hash table.

Even if a hash table restricted association pairs themselves to be immutable, the hash table itself would still be mutable because associations could be added or deleted. In this case, `EGAL` hash tables must still be identical.

The only case that would allow for consistent recursion into the structure of a hash table (as Common Lisp does with `EQUALP` [Steele90,p.108-109]) would be a constant (immutable) hash table, which represents a constant function with constant domain. In this case, the tables should be compared as extensional sets of association pairs, where the ordering of the association pairs is immaterial (as in `EQUALP`). A nonconstant function with constant domain could be simulated by such a constant hash table by binding each domain element to a different assignable cell.

The concept of a "lazy" hash table whose keys are not fully evaluated is discussed in the section on lazy values.

F. Equality of I/O Streams

There are mutable objects whose state is not changed by assignment. The most significant of these are *input/output* objects—usually input/output *streams*. One can simply define object identity through some sort of address comparison, as is done with assignable mutable objects. Such an *ad hoc* definition is not required, however. Streams are usually complex objects, incorporating a number of mutable elements such as buffers and buffer pointers, and in these cases the buffers and buffer pointers will "carry" the object's identity.

One can also conceive of unbuffered streams which are implemented as readable or writable memory locations in the manner of the "memory-mapped I/O" found in many microprocessors. For unbuffered output streams, the mutability is obvious due to the use of assignment to modify the contents of the "port". For unbuffered input streams, the "port" is read-only, but it is also *volatile* (i.e., incapable of being cached), meaning that it can change from reference to reference without ever being written. In both of these situations, the port should be considered a "mutable" cell, even if the mutating is being done by, or being noticed by, an external agent.

G. The Assignable Cell (Reference) Model

In the exposition of `EGAL`, above, we allowed for structures which have both immutable and mutable components. There is another model introduced by Algol-68 [vanWijngaarden77], however, which is cleaner because there is only one kind of mutable object—the *cell* (Algol-68 called it a "reference")—and all other objects are immutable. In the cell model, all data structures—with the exception of a cell—are immutable, and structure components which are intended to be updated are bound to cells which hold the initial value. The process of converting from assignable components to cells is called "cell introduction". We have already seen one example of cell introduction in the section on function closures, but here we uniformly introduce cells for the mutable components of all data structures.

It might seem that cell introduction would change the semantics of our `EGAL` equality predicate. This is not the case, however. While `EGAL` may have to recurse an additional level through a component, `EGAL` always stops at a cell. While the immutable structure which holds the cells together may be replicated at some point during the computation (there is no way to tell), `EGAL` will always recurse through the structure to get to the cells. The creation of an object may involve the creation of several new cells, but because the object's backbone is immutable, these cells are bound together into the structure for life. Therefore any copies of the object's immutable backbone will still hold the "same" cells. These mutable cells of an object are thus the "carriers" of a mutable object's identity, because differences in objects of the same data type can only be determined by `EGAL` recursing down to the level of cells.

If cells are the carriers of a mutable object's identity in our model, then the only way to generate nonforgeable "object ID's"—i.e., "social security numbers"—for such mutable objects is to provide a dummy cell as part of an object's definition. Since independent mutability is the essence of identity in our model, this cell can be as small as a single bit, because identity can be distinguished by "dithering" this single bit. Trivial mutable objects of this kind can be used for unforgeable keys, because the only way one could have gotten such a key is by communicating with some object which already had it, since the generator of such objects is guaranteed to produce a new, never-before-seen (modulo garbage collection) object. Of course, if the object identity predicate does not diddle this bit, and if we do not attempt to garbage collect these keys, then the storage for the bit itself can be optimized away.

In the cell model, then, all objects have the form of *finite trees* in which all of the interior nodes are immutable and all of the leaves—which can be shared—are either cells or atomic constants. All objects are finite trees, because it is impossible to create infinite trees—i.e., directed cycles—without side-effects. Furthermore, none of the interior nodes share, or more accurately, some of the internal nodes may share, but this sharing cannot be detected through side-effects or through equality testing—i.e., the tree is not re-entrant. Thus in the cell model, `EGAL` always performs a finite tree comparison. This restriction to *finite*, *acyclic* objects has significant implications for type equivalence, discussed later.

H. Object Input/Output

In Lisp, the input/output operations `READ` and `PRINT` make weak attempts to respect object identity. `READ` reads a single "object", while `PRINT` produces a `READ`-able representation of a single object. Originally, `READ` was to be a mathematical inverse of `PRINT`, so that the printed representation of an object was to be a faithful rendering of the object. When `READ` and `PRINT` are restricted to immutable cons cells, numbers and symbols, this goal is achieved—`READ`'ing such a `PRINT`'ed representation of an object A will produce an object B that is `EQUAL` to object A. Lisp I/O has never recovered, however, from the introduction of mutable objects, and `READ`/`PRINT` fail to respect object identity in the presence of mutable objects, even though some attempts are made to handle sharing and cycles.[18] Proposals for abstract I/O in other languages—e.g., [Herlihy82]—also fail to preserve object identity, because they go too far in preserving sharing for immutable objects, and do not go far enough to preserve sharing for mutable objects.

There are occasions—e.g., for Interlisp "dump" files ("poor man's persistence"), and when debugging—where the preservation of true object identity would be welcome.[19] Some sort of "social security number" is then required of every mutable object, and this number must be universal across all installations. Relational database systems which attempt to preserve object identity also require a similar "timestamp", or equivalent universal naming device. It should be clear that if every mutable object is uniquely tagged when performing an identity-preserving `PRINT`, the cost of dealing with mutable objects becomes substantially higher than the cost of dealing with immutable objects.

I. Copying

Many languages offer the ability to copy objects, including Common Lisp [Steele90,s.19.5]. Given our discussions of object identity, however, the status of such copying operations is very much in doubt. If an object is mutable, it has object identity, and copying produces a non-identical object. In other words, such a copying operation is an attempt to produce a "clone", and the depth of copying required to produce the correct semantics is highly

---

[18]Modern Lisps offer a more compact, but humanly-unreadable I/O format called "FASL" format (don't ask), which also attempts to handle sharing and cycles. FASL format also fails to respect object identity, however.

[19]The text of Autolisp interpreted user functions [Autodesk88] is often swapped to and from disk using `PRINT` and `READ`; the efficacy of this scheme depends upon the immutability of Autolisp cons cells and character strings.

application- and implementation-dependent. Therefore, no standardized copying operation (such as that for Common Lisp structures [Steele90,p.477]) will be useful.

If an object is immutable, on the other hand, every copy is identical to the original, and there is no way to distinguish among them within the programming language. Copying is therefore well-defined, because it is the identity function, but useless as a standardized function because it does nothing. Therefore, the concept of a standardized copying operation is always useless at best, and highly misleading at worst. Perhaps this ambiguity is one reason why the Common Lisp Object System (CLOS) [Steele90,s.28] has dispensed with default object copiers for each class.

An interesting situation occurs within a copying garbage collector. By definition, a copying garbage collector must preserve object identity, so the curious programmer can often find valuable insight into a programming language's notion of object identity by examining an implementation's garbage collector code. On the other hand, the collector's very copying seems to indicate a contradiction. This contradiction is resolved by the introduction of the notion of "forwarding pointers" [Baker78], which redirect object references from their original location to their new location so that the "same" object is referenced. Traditional copying garbage collectors leave forwarding pointers even for functional objects, but we have shown in this paper that forwarding pointers for functional objects are not necessary. In fact, we conjecture that 1) most recyclable garbage is functional in nature, and 2) substantial performance improvements in garbage collection algorithms can be obtained through clever optimizations for functional objects. For example, the "transitive size" of a functional object can be calculated incrementally as it is constructed, which allows any potential copier to dynamically and efficiently evaluate the trade-offs between copying and sharing. Furthermore, the reduction of the number of forwarding pointers can increase the "locality" (and hence efficiency) of a virtual memory copying garbage collector.

J. Lazy Values

Some modern languages—e.g., Scheme—offer a programmer-annotated method for achieving *lazy evaluation*. Lazy evaluation is a strategy for evaluating an expression which defers the evaluation of subexpressions as long as possible, in the hope that the value of the subexpression will never be required. For example, the evaluation of the expression denoting an argument for some call to a function may be deferred until the function itself requires the value, at which point the argument expression evaluation will be forced. Of course, the function may simply return the argument value, or store it into some data structure, in which case the evaluation may continue to be deferred. One could take the point of view that the only time that an evaluation is truly forced is when the answer is required to be printed out, but if such a printout is very large (perhaps infinite), then a lazy language may defer much of the evaluation even then. To put lazy evaluation on a more solid foundation, a lazy language usually defines a number of primitive functions as *strict*, meaning that they coerce some portion of their arguments into "ground terms".

Obviously, the uncertainty in the time of evaluation of a Scheme delay can lead to non-determinism in a language offering side-effects, as Scheme does. However, an expression in a functional language, once evaluated, will always yield the same value on additional evaluations. Therefore, most lazy implementations cache this value into a "value cell", thus avoiding the redundant evaluation of this particular expression. This use of value caches is sometimes called "call by need", whereas the uncached strategy is called "call by name".

Scheme delay creates a lazy value; (delay <exp>) creates a lazy value which can be forced into a strict value by means of the force expression, which is the left inverse of delay. In other words, (delay (force <exp>))≡(delay <exp>), but (force (delay <exp>))≡<exp>. Lazy evaluation, e.g., using delay, can be used to create infinite functional data structures, which contradicts one of our major desiderata—that egal always terminate promptly.

Prior to this section, egal has been defined as a *strict* operation, because it recursively forces its arguments until it has explored enough to determine whether they are indeed the same. The only delays left unforced will occur in arguments that are not identical, and arguments which evaluate to the same expression will be transitively forced.

If we were to implement the Scheme delay and force ourselves, by generating an assignable cache cell in the manner suggested in the Scheme document [Rees86], then we would not achieve the correct semantics unless egal strictly forced its arguments. Otherwise, (egal (delay 3) 3) would not yield true, because the function closure for the delay would not compare egal to the ground number 3. One could modify egal to be slightly less strict if it recursively forced evaluation until it compared two delays. Then, if the delays themselves were identical, which could be determined by comparing their cache cells, then egal could return true without further forcing the expression; otherwise, the delays would have to be further forced. This modification was suggested by Hughes [Hughes85] for use with "lazy memo-functions", which allow certain computations on infinite data structures to terminate.

Since our intention is the provision of a robust object identity predicate for widespread use, rather than for research programming languages, we feel that the efficiency and simplicity of a strict `egal` is more important than the transparent provision of lazy infinite data structures. Therefore, we suggest that `egal` remain strict, and those intent on implementing infinite laziness are welcome to do so using explicit assignable cells within an abstract data type. In this way, the lazy memo-functions of [Hughes85] can be readily and efficiently emulated.

4. OBJECT IDENTITY IN PERSISTENT, PARALLEL AND DISTRIBUTED SYSTEMS

Object identity becomes a much more important issue for *parallel* systems than it is in a single-process system. In a single-process system, object identity can be the same notion as that of an address, because dereferencing such an object is not expensive. With multiple processes, however, accessing an object involves some sort of locking to protect it from multiple unsynchronized accesses which could compromise its atomicity. For example, the implementation of a push-down stack in Common Lisp can utilize a vector with a "fill pointer" which keeps track of the current length of the vector. Pushing or popping such an object involves two operations—the access to the "top" of the stack and the adjustment of the fill pointer; these two operations must be grouped into an atomic transaction, or else the object will not provide correct stack semantics.

Persistent systems [Sandewall75] [Atkinson87] [Atkinson88] [Alagic89] require the logging or journaling of updates to objects, so that a consistent world can be recovered in the case of a crash. *Distributed* systems require more complexity, because in addition to having to be synchronized, an object must also be localized, and any references to this object which come from places far away from its location will be quite expensive. Although schemes have been described which enable mutable objects to be replicated in many places, these schemes are complicated and expensive; therefore, we will consider only models in which each mutable object has but one location.

We have argued that immutable functional objects have "object identity", but they do not require synchronization because they cannot be modified, and they do not require localization, because they can be replicated at will. As a result, functional objects can be much more efficient than non-functional objects in a distributed system; this is the reason for Cedar's functional strings (called "ropes") [Swinehart86]. For example, if we were to create a list of objects in a distributed Lisp system, traditional Lisp semantics would demand that each list cell have "object identity", because Lisp list cells are mutable. As a result, a function call from one location to another would pass a pointer to this list, and the called function would then have to interrogate each list cell in turn (perhaps using a locking protocol) to acquire references to the listed objects. If, however, our Lisp had functional list cells, we could pass this list by copying it at the time of the function call (in the manner of a "remote procedure call"), and the called function would then have its own private copy of this list after only one interaction with the calling function instead of many. Keeping consistent semantics in this situation is possible only if argument lists are functional.

The efficiency of functional arguments in parallel and distributed systems does not come from either a consistent reference-passing or a consistent copying policy, but from the freedom to do *either* at any time. The consistent reference policy allows for great efficiency for infrequently referenced or very large objects, while the consistent copying policy allows for great efficiency for frequently referenced and/or small objects. By basing object identity upon an abstract concept like mutability instead of an implementation-dependent concept, we can achieve efficiency in a more portable fashion.

Ada's non-deterministic, non-transparent semantics (by-reference v. copying) for its "**in out**" parameter-passing mode has the same efficiency goals as our object-identity proposal; Ada, however, must rely on the good intentions of the programmer, because the difference between the two mechanisms is all too evident. Since "the road to hell is paved with good intentions", our scheme does not rely upon them for correct behavior, unlike Ada's "**in out**" parameter-passing.

5. OBJECT IDENTITY IN MULTILINGUAL ENVIRONMENTS

The ability of a program in a language to call subprograms in other languages has become an feature essential to a programming language's ability to survive the recent "standardization craze". The mechanisms required for interlingual communication are quite similar to those required for heterogeneous distributed systems—even when the various languages run within the same address space.

Communication *per se* in a multilingual single-address-space environment is not as big of a problem as protecting one language environment from the other. The differing structure formats of different languages can usually be handled by relatively straight-forward format coercion routines. Much more problematical is the fight between the language run-time systems over the control of storage allocation and control of object identity.

Functional argument structures can reduce interlingual communication costs, just as they do in a distributed environment. Copies of functional objects can be freely made using the recipient language's own storage management system, if necessary, so that no storage management problems arise; functional return values are handled similarly. Of course, if the functional objects are small enough, then no storage need be allocated.

In the case of mutable objects, it is very likely that the different languages have different formats and thus the receiver of an object cannot access it directly. In this case, the sending language can pass the object after it has been encased in a translation container which interprets actions requested by the receiver. This translation container is essentially a function closure or an Algol-60-like "thunk". Thus, even when a mutable object must be passed, the existence of cheap functional objects—argument lists and closures—can still reduce the overall costs.

Unfortunately, the publication of object references creates a substantial "garbage collection" problem in distributed and heterogeneous environments. Since the owner of the object is never sure when the last object reference has been forgotten, he is obliged to keep the referenced object alive indefinitely. We have no elegant solution for this problem, but the aggressive replacement of mutable objects by functional objects can reduce the amount of "persistent garbage" that is created by interlingual communication.

6. MOSTLY FUNCTIONAL PROGRAMMING

Due to the growing importance of persistent, parallel and distributed programming systems, and due to the costs of mutable objects in such systems, "functional" and "mostly functional" programming styles have been advocated [Backus78] [Knight86]. There are numerous advantages to eliminating gratuitous side-effects. Not only do we gain in efficiency in persistent, parallel and distributed systems, but we also make the job of compilers and verifiers much easier [Baker90d]. Less obviously, the freedom to copy functional objects or not can improve the underlying operations of storage management [Baker78] [Baker92a]; for example, *forwarding pointers* are not required when a functional object is moved because the sharing of functional objects is not detectable by `EGAL`.

Providing for mostly functional programming in Common Lisp requires the addition of functional list cells and functional arrays. Such provisions simplify many complications of Common Lisp. There are many rules regarding the sharing/non-sharing/modifiability of constants in source code;[20] these rules disappear with the single policy that traditional parenthesized lists and quoted strings `READ` as functional objects. Making `CONS` and `LIST` produce functional list cells, and including a new constructor (`WCONS`?) for mutable list cells,[21] offers the efficiency of functional lists to the 95+% of instances where lists are used functionally. Of course, `EVAL` should accept source code only in the form of functional lists; modifiable source code has been considered an anathema for the past 30 years. The problem of whether `&REST` arguments in argument "lists" are copied or not disappears when `&REST` arguments become functional.

Functional strings can have interesting implementations. A representation of a string as a tree whose leaves are individual characters offers O(1) complexity concatenation; while element indexing is more expensive, it is usually less common than concatenation.

Functional arrays are known to have performance problems, because a succession of single-element updates may convert a problem of linear complexity into one of quadratic complexity [Bloss89]. Many solutions to these problems have been proposed [Schwartz75ab] [Hudak85] [Hudak86] [Hudak89] [Bloss89] [Baker90a] [Baker91a]. On the other hand, functional array semantics can allow the experimentation with completely new array implementations, such as quad-trees [Wise87]. The incorporation of functional arrays into existing programming languages would allow for the extensive testing of these ideas, while preserving the option of traditional mutable arrays.

The status of primitive similarity predicates that are coarser than `EGAL` is marginal in a mostly functional language. For example, suppose we wish to compare a mutable character buffer with a constant string. Since these two objects have different types, there is no obvious mechanism to compare the two, and they might have different representations. The language could offer a primitive coercion from a mutable character vector into a functional string, which could then be compared for equality. Any other similarity primitive would be an attempt to optimize this two-step process.

7. CALL-BY-REFERENCE VERSUS CALL-BY-VALUE

Despite the thirty years that the problem of "call-by-reference" versus "call-by-value" in argument-passing semantics of Algol-like programming languages has been discussed, it has never been clearly resolved. While many undergraduate texts [Aho86,s.7.5] and language manuals give a definition of each policy, there has always been a nagging feeling that these definitions for Algol-60, Fortran and Ada are too *ad hoc* and too implementation-oriented.

---

[20]This is called the "coalescing of constants" problem [Steele90,p.691-4].

[21]Given the availability of `defstruct`, perhaps primitive mutable list cells are obsolete.

With the proliferation of persistent, parallel and distributed systems (with, e.g., "remote procedure calls"), the problem of resolving the precise definition of argument-passing semantics has become urgent.

There are two orthogonal issues in argument-passing semantics: evaluation ordering (e.g., "eager" versus "lazy"), and "object identity". Unfortunately, these two issues interact in very complex ways in non-functional languages, and this interaction has produced a great deal of confusion. Evaluation ordering is well-defined and relatively well-understood, because it can be studied in the context of the functional lambda calculus which has well-defined and precise semantics. However, the "object identity" issues have not been well-understood, precisely because they are intimately tied up with the side-effects introduced in non-functional languages.

We claim that the only argument-passing model that is consistent in non-functional languages is *call-by-object-reference*, i.e., passing object identities. Exactly *when* this object identity is determined relative to the computation inside the called function is the issue of evaluation order, but *what* this object identity means should always be well-defined. Our model offers a seamless integration of functional and imperative programming, because if an argument is functional, it will be passed by value, and if it is mutable, it will be passed by "reference". Any other argument-passing model involves an implicit coercion of mutable objects or mutable components into their values, and the timing of this coercion may be difficult to determine or control. For example, the coercion semantics of `EQUAL` applied to standard, mutable list cells is best described as "lazy coercion to value", where only the portion of the argument examined is actually "coerced". The complexity of such gradual coercions is one of the main reasons for our rejection of the traditional argument-passing approach.

Lisp is sometimes characterized as "call-by-value", but this is incorrect. Lisp approximates call-by-object-reference semantics extremely well, and the single lacuna can be easily repaired. The reason Lisp appears to be call-by-value is that arguments which are functional data structures appear to have been passed in toto. However, it is the nature of functional data structures that whether they are passed by value or by reference is not determinable, hence we consider these data structures to have been passed by "object reference".

"Call-by-value" coerces arguments into their "values", which causes mutable portions of a structure to be replaced by their current values; Lisp does not usually do this. The single exception is caused by variables, which are made variable by `SETQ`. If we make variable binding permanent, then we can eliminate this exception. `SETQ` of global variables is trivially replaced by `SET` of the quoted variable, while `SETQ` of local variables can be eliminated through "cell introduction" for mutable local variables, discussed in several sections above. After these changes, the binding of all lambda variables is constant (although they are sometimes bound to mutable cells), and argument-passing becomes completely uniform. Of course, when cells are introduced for mutable lambda variables, any variable instances—including those in argument lists—are replaced by expressions which access the cell's value, thus making any coercion explicit.

8. TYPE IDENTITY, TYPE EQUIVALENCE AND ABSTRACT DATA TYPES

A. Comparing Type Objects

There has been as much confusion over type identity as there has been over object identity, although the type identity problem is usually referred to as the type *equivalence* problem [Aho86,s.6.3] [Wegbreit74] [Welsh77]. The type identity problem is to determine when two types are equal, so that type checking can be done in a programming language.[22] Algol-68 takes the point of view of "structural" equivalence, in which nonrecursive types that are built up from primitive types using the same type constructors in the same order should compare equal, while Ada takes the point of view of "name" equivalence, in which types are equivalent if and only if they have the same name. We will ignore the software engineering issues of which kind of type equivalence makes for better-engineered programs, and focus on the basic issue of type equivalence itself. We note that if a type system offers the type `TYPE`—i.e., it offers first-class representations of types as accessible objects during execution, then the type identity problem is subsumed by the object identity problem [Goldberg83] [Queinnec88].

Type systems are usually constructed starting from a small set of primitive types—boolean, character, IEEE-single-float—and building up utilizing *products* (record structures and arrays), *sums* (*unions*) and *functions*. Any *finite* type expression built in this way can be "easily" compared for type equivalence using structural equivalence, although comparing unordered union types with less than quadratic complexity requires some sophistication. Equivalence problems arise in two areas: recursive types and abstract data types. Recursive types cause problems because their proper description requires either infinite type expressions, or type expressions with directed cycles [Aho86,s.6.3]. In either case, the obvious simple recursive structural equivalence algorithm will fail. Abstract data types cause

[22] Whether type checking is performed statically during compilation or dynamically during execution is irrelevant for our purposes.

problems because they are supposed to be *opaque*; i.e., equivalence should depend only upon the identity of the abstract data type itself, and not upon whatever representation is currently being used to implement it [Guttag77].

Regardless of their theoretical problems, recursive data types are handled in the following way by Pascal, C, Ada, etc. The programmer first declares that the recursive type is a type (we have purposely fuzzed the distinction between "type" and "type name"), but gives no details; then he uses the type to construct another type. Finally, he gives the full definition of the recursive type.[23] Inside of the Pascal, C, Ada, etc., compiler, it is obvious what is happening. A new type node is defined, but its contents are not filled in. This type node is referenced by one or more other type nodes. Finally, the type node is *updated* with the full definition of the type. We notice two things about this implementation of recursive types: a recursive type involves a true directed cycle in the type definition data structure, and building a recursive type involves a side-effect (assignment) on the type definition data structure. Directed cycles cannot be built without side-effects, and conversely, side-effects are only needed to build these cycles, because non-cyclic types can be built in a functional manner.

Abstract data types are handled in a similar way by Eiffel, Ada, etc.[24] The programmer first declares the *specification* of the data type, which involves giving it a name.[25] Later in the compilation, linking or execution, the datatype is associated with an *implementation*, which provides a representation for the datatype. Since a user of the abstract data type is not allowed to know the implementation of the datatype, because it is opaque, it cannot know its representation, and therefore it cannot perform structural equivalence. Inside the Eiffel, Ada, etc., compiler, it is obvious what is happening. A new type node is defined, but its contents are not filled in. This type node is referenced by one or more other type nodes. Finally, the type node is *updated* with the full definition of the type when the implementation is given. We notice that an abstract data type seems to involve a side-effect on the type system when the implementation is finally given, and this side-effect seems to be inherent in the notion of abstract data type, since abstract data types can be given new implementations.[26]

We make the claim that recursive data types for persistent languages with the type `TYPE` are most naturally defined as abstract data types whose *implementation* depends upon the abstract data type itself. In other words, since we wish to provide abstract data types anyway, and since the abstract data type provides us with the side-effect we need to achieve recursive type circularity, we might as well enjoy the conceptual simplicity this scheme provides. There is then no need for complex structural equivalence algorithms for recursive types, as is required for Algol-68, because the cycles in all recursive types are now broken by an abstract data type which must compare identically. Furthermore, the intermediate states involved in building a recursive data type are usually visible from a parallel process in a parallel/distributed/persistent system having explicit type objects, in which case the type will display its mutability, and therefore may as well be an abstract data type.[27]

In summary, we have described a type system which provides a small set of primitive immutable types (*boolean*, *bit*, etc.), a small set of primitive type constructors (*product*, *sum*, *function*, etc.), an operation to create a new, mutable opaque type (not to be confused with a Milner-style type *variable* [Milner78]), and an operation to assign a mutable type a type-expression value. Given this type system, the type identity/type equivalence problem is solved by the straight-forward application of our `EGAL` equality predicate. Type values are finite trees which are constructed from leaves which are either immutable atomic types (*boolean*, *bit*, etc.) or mutable opaque types. The backbones of these trees are immutable structures (*product*, *sum*, etc.). `EGAL` performs structural equivalence down to the leaves, which compare for identity.

The difference between languages with structural type equivalence and languages with name type equivalence is now clear. Languages with structural type equivalence (e.g., Algol-68) use a permanent binding of names to type values, so that the type comparison recurses through the type name. Languages with name type equivalence (e.g., Ada) use

---

[23]In simple cases these steps may be performed within a single type definition, but these rarely occur [Baker80].

[24]We purposely ignore the Common Lisp Object System [Steele90,ch.28], because the types it introduces are not completely opaque, and hence they are not true abstract data types. For example, the slot structure of a type can be ascertained by examining its "class precedence list".

[25]The new data type itself is the key issue for us; we ignore any operation signatures also given that refer to the new datatype.

[26]This updating involves a bit of checking to make sure that the operation signatures match, but these details do not change the side-effect nature of providing an implementation for an abstract data type.

[27]In an analogous situation, the internal side-effects involved in the implementation of Scheme's `letrec` can be exposed using continuations, allowing a functional subset of Scheme to emulate side-effectable cells! [Bawden88].

an assignable binding of names to type values, so that the type comparison stops at the type name. By forcing recursive types to be abstract data types, Ada could dispense with a whole set of rules and restrictions for "incomplete types", which are used in Ada to construct recursive types.

Our assertion that type identity is intimately bound up with the mutability of an abstract data type is essentially a *modal* concept. Modal logic [Hughes68] deals with the notions of "possibility" and "necessity", and Kripke models of these logics utilize the notions of "multiple worlds", where "possibility" is intimately related to "accessibility" of other worlds. A data type is thus abstract if we can "conceive of" multiple implementations of it; i.e., if there exist worlds accessible from the current world in which the implementation differs. In modal "dynamic logic" of programs [Pratt79], accessibility is achieved through one or more assignment statements, so it should be possible to unite these two concepts. This modal notion of abstract data types is closely related to the "existential" notion of abstract data type espoused by [Cardelli85] and [Mitchell88].

B. Comparing Objects of Abstract Data Types

Abstract data types [Guttag77] are so-called because they present the specification of a set of objects which is abstracted from the representation of those objects. It is therefore important to distinguish between the external (specificational) view of an abstract object and the internal (representational) view of the object. The simplest treatment of object identity for abstract objects is to ignore the abstraction and use the object identity of the representation; this choice is the only one available to programmers in languages which do not support abstract data types. This simple treatment has two drawbacks—1) it allows the equivalencing of two objects which *accidentally* have the same representation—e.g., the rational number 2/3 and the Gaussian integer 2+3*i*; and 2) it requires that the abstract data type representations always be kept in *canonical form*—e.g., Ada has this problem because its "=" operator is not easily overloaded with any other interpretation. For example, the comparison of (functional) rational numbers would be performed by comparing their numerators and denominators; this comparison would only work correctly for pairs with no common divisors in which the sign appeared uniformly in either the numerator or the denominator. When a canonic form exists, it can greatly improve the efficiency and reliability of the overall program, because the equality predicate is under the complete control of the implementation and hence optimized and trusted. Unfortunately, some abstractions may not have a canonical form, or else continual canonicalization may be inefficient. In these cases, the programmer may wish to offer an *abstract object identity* for objects of the abstract data type which is different from the representational object identity; he will then need to overload (genericize) the `EGAL` predicate.

`EGAL` avoids the confusion between abstract and concrete objects by its first clause which requires that the datatypes themselves be `EGAL`. Since an abstract datatype itself includes a mutable cell which carries the type's identity, it is this cell which is compared when abstract datatypes are compared. An object of the abstract type can be "opened up" to reveal its *representation*, which is another object of another—usually concrete—type. By requiring the conjunction of abstract type equivalence and concrete representation equivalence, `EGAL` will distinguish between abstract objects which may be abstractly equivalent, but have different representations—perhaps as the result of a new implementation of the abstract type having been instantiated. Although this equivalence may be finer than we might desire, at least it will not lead to contradictions.

```
(defun egal-abstract-type (x y)
  (and (egal (type-of x) (type-of y))
       (egal (representation x) (representation y))))
```

We would now like to be able to overload `EGAL` for abstract data types in a way that provides a more abstract equality, but does not compromise `EGAL`'s basic principles. If the proper overloading of `EGAL` can be automatically derived from the other information given in defining the abstract data type, then `EGAL`'s nature will be preserved. We can derive the requirements for abstract `EGAL` through a number of observations.

Rule I. Two abstract objects can be `EGAL` only if they have the same (abstract) data type.

Rule II. `EGAL` operating on abstract objects (abstract `EGAL`) cannot be finer than `EGAL` operating on their representations (representational `EGAL`). This rule also guarantees that abstract `EGAL` is reflexive.

Rule III. Abstract `EGAL` must be symmetric. A smart compiler could check this condition at compile time by comparing the given user code to the user code with its parameters reversed; if the two are equivalent, then the user code is symmetric.

Rule IV. Abstract `EGAL` must be transitive. A smart compiler could check this condition at compile time by noticing when the abstraction calls another predicate which is already known to be transitive—e.g., if it calls `EGAL` on component values.

Some of the above rules can be easily enforced by inserting any user definition for EGAL into the following prototype:

```
(defun abstract-egal (x y)
  (and (egal (type-of x) (type-of y))            ; Rule I.
   (or (egal (representation x)
             (representation y))                 ; Rule II.
    (let ((user-egal (get-egal (type-of x))))
      (or (funcall user-egal x y)
          (funcall user-egal y x))[28]            ; Rule III.
       ...))))                                   ; other Rules.
```

Unfortunately, the requirement for transitivity cannot be so easily guaranteed. For example, the transitivity of the grade-school cross-product rule for comparing rational numbers depends upon deep properties of integer multiplication—associativity, commutivity and cancellation. If one is preparing "mission-critical" or "standards-quality" abstract data types, it might be reasonable to ask for a mathematical proof of transitivity before allowing the compilation of a user EGAL definition. Under normal conditions, however, this would be impractical. Short of a mathematical proof, we can achieve some assurance from the prototype above; we believe this approach preferable to the approach of many "object-oriented" languages which allow any behavior of an identity predicate whatsoever.[29]

An even more difficult case is that of an abstract object whose specification claims the object to be immutable, but whose representation utilizes mutable objects and *benevolent* side-effects. The most obvious example is that of the use of a reference to RAM storage to represent a functional object (e.g., a Lisp *bignum*); traditional abstract datatype systems handle this situation by protecting the object from outside access either intentionally, through the published operations, or unintentionally, by clobbering storage through wild pointers. A more sophisticated example is the use of benevolent side-effects to improve the performance of functional arrays through *shallow binding* [Baker91b] (also called *trailers* [Bloss89]). If the implementation is flawed in some way that allows the internal state to become visible, then havoc will result. Proofs of immutable behavior are even more difficult than in the rational number case, above, although work is progressing on type systems that can automatically check simple cases [Gifford86] [Lucassen87].

As a result of these problems, we recommend that EGAL remain a simple, reliable, decidable predicate which compares types and representations, as in the following code. While EGAL will sometimes provide a finer equivalence relation than the abstract one desired, it cannot cause embarrassment due to violation of the basic rules of object identity, because it tests for identity as preserved by the basic value transmission operations of the programming language.. Furthermore, this EGAL is consistent with the earlier definition of EGAL on closures, which are often used to implement abstract data types (this consistency can be seen by equating the notion of an abstract type object with the notion of the code for a closure). Systems defining abstract datatypes might consider providing a new "generic" predicate that defaults to EGAL for primitive datatypes and can be overloaded for user-defined abstract datatypes; this generic predicate would then be used for table lookup routines such as `assoc`, `member`, etc., instead of passing an equality predicate as an argument.

```
(defun egal (x y)
  (and (egal (type-of x) (type-of y))
    (cond ((abstract-type-p (type-of x))
           (egal (representation x) (representation y)))
          << the other clauses for egal, as before. >>
           )))
```

9. COMPARISON WITH PREVIOUS WORK

Philosophers and logicians—including Aristotle—have long discoursed on the *properties* and *attributes* of *objects*. Von Wright, according to [Hughes68,p.184], distinguishes between *formal* properties, "whose belonging to an object is always either necessary or impossible", and *material* properties "whose belonging to an object is always contingent". If we interpret "necessary" and "possible" according to the modal systems of "dynamic logic", which

---

[28]A smart compiler can often optimize out this second call to `user-egal` which guarantees symmetry.

[29]We note that this EGAL prototype does not conform to the usual object-oriented dispatching mechanisms (e.g., Smalltalk and Common Lisp Object System (CLOS) [Steele90,ch.28]), because we do not call the specific user function first. In CLOS, EGAL would be a non-generic function which eventually called the specific user function.

models imperative sequencing and assignment [Pratt79], then formal properties are equivalent to functional properties (components) of an object while material properties are equivalent to mutable properties (components) of an object. Obviously, material properties—being contingent—cannot contribute to an object's identity, while formal properties—being necessary—are an inherent part of an object's identity.

McCarthy's *Lisp* language[McCarthy60] clearly demonstrated his understanding of side-effect semantics for object identity through his use of "mark bits" in a mark-sweep garbage collector.[30] His `EQ` function was restricted to work only on *atomic* (non-structured objects), and his recursive `EQUAL` function worked correctly for all objects because his cons cells were immutable. His followers forgot to re-examine the situation, however, once `RPLACA` and `RPLACD` had been introduced into Lisp, perhaps for reasons of compatibility.

[Steele78,39-43] came tantalizingly close to drawing our conclusion: "The concept of side effect is inseparable from the notion of equality/identity/sameness", and later: "the only way one can determine that two objects are the same is to perform a side effect on one and look for an appropriate change in the behavior of the other". This quote makes inexplicable the later conclusion of the ANSI Common Lisp committee, of which Steele was a member, that "object equality is not a concept for which there is a uniquely determined correct algorithm" [Steele90,p.109].[31] The radical consequences of the obvious conclusion were apparently too much for either Steele or the ANSI committee to accept—that there were *two* different kinds of list cells, mutable and immutable, with coercions between them, and recognizing this might have sacrificed upwards compatibility.

Common Lisp [Steele84] [Steele90] defines the predicates `EQ`, `EQL`, `EQUAL`, `EQUALP`, which are listed in fine-to-coarse order. In addition, Common Lisp offers `"="`, which is `EQUALP` restricted to numbers. `EQ` essentially compares pointers, while `EQL` relaxes `EQ` and allows functional objects like characters and numbers to be correctly compared.[32] `EQUAL` structurally compares conses, strings and bit-vectors, while `EQUALP` structurally compares all vectors and fuzzes upper/lowercase character distinctions. Unfortunately, Common Lisp arrays cannot be defined as "read-only", so functional arrays (including functional strings and bit-vectors) cannot be defined. Common Lisp does allow for "read-only" components of structures, but `EQUAL` does not decompose structures and `EQUALP` decomposes all structures.

Scheme [Rees86] defines the predicates `eq?`, `eqv?` and `equal?`. `eq?` is essentially pointer comparison (Common Lisp's `EQ`), while `eqv?` is a crude attempt at achieving "operational equivalence", which is not at all well-defined; `eqv?` is approximately the same as Common Lisp's `EQL`, except on functional closures. Finally, `equal?` is Scheme's approximation to Common Lisp's `EQUALP`, since it approximates structural equivalence. While Scheme allows for `eqv?` to implement our version of `EGAL`, it does not require it, and we are not aware of any Scheme which follows our semantics.[33]

Goto [Goto74] [Goto76] researched the notion of "hash consing" (invented by Ershov for common subexpression detection [Ershov58]) which provided an efficient implementation of `EQUAL` for *functional* cons cells. However, we are more interested here in the fact that he provides a separate datatype for functional cons cells than in the fact that they can be efficiently compared. He provides the same semantics for `EQ` that we do for `EGAL`, but accomplishes this by forcing the "uniquizing" of functional cons cells so that the existing `EQ` works correctly. We, on the other hand, do not constrain the implementation of `EGAL`, but allow `EGAL` to handle non-uniquized functional cons cells.

MacLisp [Moon74] was the first Lisp to provide stack-allocated numbers which became "first-class", when they escaped the boundaries of dynamic extents [Steele77]. As a result, `EQ` for numbers became problematical, since numbers could be moved without notice, and these techniques resulted in Common Lisp's `EQL` predicate. This paper can be considered a generalization of this `EQL` technique to all functional objects. The generalization of the stack-allocation techniques of Maclisp to arbitrary objects is considered in [Baker92a]. MacLisp also had a function

---

[30] The idea of mark-sweep garbage collection is at least 4,000 years old; the Jewish holiday *Passover* celebrates the effectiveness of the first distributed marking algorithm against a particularly deadly sweeping algorithm [*Exodus*12:23-27].

[31] The recent incorporation of CLOS (Common Lisp Object System) into Common Lisp requires the precise definition of object identity, because CLOS generic functions can "dispatch" on particular objects (using `EQL`).

[32] [Moon89] suggests that "the Common Lisp function `EQL` compares two object references and returns *true* if they refer to the same object". While Moon's statement is usually correct, most people mistakenly read the "if" as though it said "if and only if".

[33] Curiously, [Rees86] defines operational equivalence for *mutable* structured objects, but not for *immutable* (functional) structured objects, and hence operational equivalence is ill-defined.

`PURCOPY`, which coerced its argument into an immutable form for sharing among processes in a time-sharing system. MacLisp had no predicates to distinguish pure from impure objects, however.

AutoLISP [Autodesk88] is the only Lisp we know of which offers only functional strings and functional cons cells. Unfortunately, it is not pure, because it offers `SETQ`. String and cons cell functionality is not the result of theoretical considerations, but the result of swapping objects to disk using `PRINT` and `READ`. This is because any side-effects (e.g., sharing) to non-atoms are forgotten when the object is swapped in.

Parallelizing Lisp has been the subject of much research. [Baker77] introduces the concepts of *eager* evaluation and *futures*; *MultiLisp* [Halstead84] [Halstead85] implements and studies these concepts in great detail. *ParaTran* [Katz86] [Tinker88] utilizes the ideas of "time warp" to synchronize sequential Scheme on a parallel processor. *QLisp* [Gabriel84] introduces futures into Common Lisp. [Larus89] investigates aliasing in Lisp data structures, while [Harrison88] suggests new—more functional—list structures for Lisp in a parallel processing environment. Parallel Lisp constructs for SIMD architectures are considered in [Hillis85] and [Steele86].

Smalltalk [Goldberg83] [Digitalk88] has two equality predicates—"==", for "object identity", and "=", for "equality". Smalltalk's "`==`" is roughly Lisp's `EQ`, while Smalltalk's "`=`" (on built-in classes) is roughly Lisp's `EQUALP`, which descends into objects regardless of their mutability. Smalltalk intends "=" to be an equivalence relation coarser than "==", but a programmer can define "=" to mean anything at all, including "`not ==`". Smalltalk has no notion of functional objects which can be compared extensionally.

Prolog [Warren77] implicitly assumes the existence of object identity and an object identity predicate for the operation of its unification algorithm. Since most Prologs are "pure"—no side-effects to data structures—the proper definition of object identity has never been an issue. However, recent attempts to integrate Prolog-like mechanisms into (impure) Lisp [Robinson82] and Scheme [Ruf89] require a more precise notion of object identity; e.g., [Ruf89] incorrectly uses `eq?` rather than `eqv?`.

Algol-68 [vanWijngaarden77] is apparently the first standard language to rigorously separate the concepts of "value" and "name" (" mode ref" ≈ "assignable cell"), which model has been followed most closely by ML. Algol-68 provides the object identity predicate ":=:" only for two name/refs, which compares the names themselves for identity, not the currently-assigned values [vanWijngaarden77,5.2.2]; any attempt to compare a name to a value yields a compile-time error, since a name/ref does not have the same mode/type as a value. The "=" operator is initially overloaded only for the arithmetic, boolean and character types, but the programmer must overload "=" himself for any other type. Algol-68 does not have abstract data types, but does allow for recursively defined types which must be extensionally compared, thereby producing the complexity traditionally associated with Algol-68 type equivalence.

CLU [Liskov77] has a concept of object identity which is similar to ours, in that it distinguishes mutable from immutable objects. CLU defines mutability relative to the operations supplied by the programmer of an abstract data type, however, making immutability substantially more difficult to prove [Bloom76]. CLU also allows for the definition of recursively defined types "without explicit reference types" [Liskov77]; however, the example given in that paper contradicts this assertion through its use of assignment. Herlihy [Herlihy82] describes a mechanism for changing the representation of an abstract "value" during transmission in CLU, and uses the notion of "value equality" to determine the fidelity of this transmission. Unfortunately, the definition of value equality is left up to the particular abstract datatype, and this predicate does not have to be an equivalence relation—e.g., his example of floating-point numbers and rectangular/polar complex numbers uses *closeness* instead of *equality*. While Herlihy uses the term "call-by-value", he actually describes "call-by-object-reference" (although he uses the term "name" instead of "object reference"), because true call-by-value wouldn't care about sharing and cycles. Finally, his attempts to preserve sharing and cycles are not well-defined, because he does not acknowledge the mutability of "names", even though his implementation works exactly like a copying garbage collector, including the updating of forwarding pointers stored in the name-to-object "maps". Since the object identity of these "names" is not preserved outside a single run of a particular program, he underestimates the real costs of full object identity in a persistent database.

ML [Harper86] has a notion of equality very similar to ours. "On references [assignable cells], equality means identity; on objects of other types ..., it is defined recursively in the natural way" [Harper86,s.7.2]. ML makes no attempt to compare function closures, however, even when they simulate data structures. Ohori [Ohori90] makes the case that ML references (cells) are the most appropriate means for introducing object identity into a pure functional language.

C [ANSI-C] has only a single predicate "==" which acts like Common Lisp `EQ`. C "==" requires the objects being compared to be either arithmetic values (including character values) or pointers. C accesses arrays only through pointers, so C "==" does not descend into the contents of arrays. Since C does not define "==" for structures, it finesses the issue of how best to compare them. C approximates the notion of a functional object through the "`const noalias`" declaration, which can be applied to structure components; C "`==`" does not cope with this declaration, however, because pointer comparison compares addresses. *C++* [Stroustrup86] follows C, but allows "`==`" to be overloaded. There are no requirements on user-supplied definitions for "`==`"; in fact, "`!=`" could be defined as equality and vice versa.

Ada [Ada83] has a single predicate "`=`" which acts like Common Lisp `EQUALP` because it recurses on the components of structures and the elements of arrays. Ada "`=`" does not descend through pointers, however, so it cannot run into the possible circularity of `EQUALP`. Object identity for accessed objects is determined by address, which is consistent, because Ada does not allow the declaration of constant (non-assignable) aggregate components. Since "`=`" compares the contents of aggregate variable components, "`=`" is not referentially transparent. "`=`" in Ada is not easily overloaded, and since the functionality of an aggregate structure is not encoded in its type and cannot be determined by the program at run-time, it is difficult to implement our `EGAL` semantics.

While one cannot declare a component of an Ada object immutable ("constant"), one can declare an "entire" object constant. Unfortunately, if one compares two pointers to such constant objects, the pointers are not dereferenced in the comparison—i.e., they are compared intensionally instead of extensionally. In a noble attempt at cleanliness, Ada restricts formal mode "**in**" parameters of subprograms to be "read-only". This restriction is an attempt to convey the notion that arguments are coerced into immutable values before being passed as arguments. Unfortunately, this restriction is independent of the arguments, and does not extend to pointer-dereferencing. Ada's read-only parameters thus cause more confusion than they resolve.

Ada retains the notion of "mode" from Algol-68, but separates it from the notion of "type". Ada modes are orthogonal to Ada types, and apply only to parameter-passing. Although one can utilize Ada's "**in out**" mode to partially preserve object identity for *variables* (Ada's mutable objects) [Baker91b], object identity in Ada is generally preserved only for "access" types—i.e., Algol-68/ML *references*, i.e., *pointers*. Much of Ada's complexity in semantics and implementation is a result of its unwise decision to separate modes from types, and from its attempt to emulate "cache coherency" itself in software using mode "**in out**".

Hewitt's *Actor* systems [Atkinson77] [Hewitt78] [Agha86] incorporate the notion of "serialized" and "unserialized" actors. Actors with state are serialized, while functional actors need not be serialized. Serialized actors and unserialized actors correspond roughly to objects with "object identity" and functional objects, respectively. The discrete linear time ordering required of any actor with internal state [Hewitt78] is essence of "serializability", required for (cache) coherence.

MacLennan [MacLennan82] [MacLennan85] independently provides arguments quite similar to ours for a notion of object identity essentially the same as ours. He does not, however, consider the equality problem for function closures.

Gifford [Gifford86] has been examining the inclusion of side-effect information into type systems. He has not examined the issue of object identity, but his type systems should be capable of providing the information necessary to implement our `EGAL` predicate.

Linda [Carriero89] utilizes a pattern-matched database of structured values ("tuples") for communication among independent processes in an MIMD parallel processing environment. A Linda *tuple* is a functional data structure which can have additional structures as components. Tuples are not recursive and are compared recursively—i.e., tuples are finite functional objects, and Linda has no mutable objects other than the global database itself.

Relational data base theory has the concepts of "key", "primary key", "functional dependency" and "normalization". Functional dependencies can be operationally exposed through "update anomalies" which occur when functional dependencies are violated. Update anomalies can be minimized through normalization, which factors the relations according to their functional dependencies. Normalization attempts to whittle an object's identity down to its "primary key", while we grow the object identity to include all of its functional dependencies. Immutable relations cannot be updated, however, and therefore cannot be used to detect update anomalies and functional dependencies. Since these functional dependencies can all be viewed as immutable attributes of the object (whether direct attributes or transitive attributes dependent upon an immutable direct attribute), the inclusion of all of these attributes as part of the object cannot cause an inconsistency. The essence of our argument, when rephrased in relational data base terminology, is that whether immutable relations are normalized can't be determined through update anomalies, and hence can't matter.

Vianu's conception of "object identity" [Vianu88] seems identical to ours, although it is couched in the form of relational algebra rather than in the form of object-oriented programming. Earlier work [Lyngbaek87], however, separated the world into "literal" (i.e., functional/immutable) objects and "non-literal" (i.e., objects with "true" object identity).

10. CONCLUSIONS

Although efficient object-oriented applications can be designed and built in any language, clean semantics for object identity are required in order to eliminate the need for heroic measures [Baker91c]. We have shown that object identity is best defined by the transitive closure of immutable attributes of an object. Object identity is thus defined by side-effect semantics in a way analogous to the way that normal forms in relational algebra are defined by update anomalies.

Our discussions have revealed the `EQ`/`EQUAL` problem to be a straight-forward typing error. Applying `EQ` to a functional list or applying `EQUAL` to a mutable list are type errors of the same sort as applying a floating-point equality predicate to two integers. That mutability has not previously been considered an important issue in programming language typing must be the result of an artificially low price on the assignment operation in von Neumann architectures.

Our primitive `EGAL` predicate is the coarsest predicate consistent with object identity, and therefore `EGAL` should be the finest equality predicate primitive in a language. In this way, many inconsistencies and anomalies disappear, including some especially troubling ones involving hash tables and property lists. Our notion of object identity provides a firm foundation for the introduction of immutable list cells, immutable arrays and strings, and immutable structures into modern programming languages, including Common Lisp. These immutable objects immediately solve many problems which have troubled language standards committees; for example, many of the problems of "typing for declaration" and "typing for discrimination" [Steele90,p.53] stem from imprecise object identity. Immutable objects can also lead to improved performance in persistent, parallel and distributed execution environments.

Our notion of object identity provides for a clean semantics of argument-passing and value-returning, eliminating many *ad hoc* and confusing rules with a single notion—*call-by-object-reference*. Call-by-object-reference is precise, and is robust when used in persistent, parallel and distributed systems.

By providing a better understanding of object identity and its costs, we hope to advance the cause of "mostly functional programming" [Knight86], which tries to reduce the number of "gratuitous" side-effects. Mostly functional programming results in higher efficiencies in persistent, parallel and distributed processing environments because logging, communication and synchronization costs are reduced.

ACKNOWLEDGEMENTS

Many thanks to Matthias Felleisen, Dan Friedman, Robert Kessler, André van Meulebrouck, and Carolyn Talcott for their comments on early drafts of this paper. We thank the referees for bringing the details of object identity in Algol-68 and CLU to our attention, and for their other helpful suggestions.

REFERENCES

Abelson, H., & Sussman, G.J.. *Structure and Interpretation of Computer Programs*. MIT Press, Camb., MA, 1985.

Abiteboul, Serge, and Kanellakis, Paris C. "Object Identity as a Query Language Primitive". *Proc. 1989 ACM SIGMOD Conf., Sigmod Record 18*,2 (June 1989),159-173.

AdaLRM: *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, U.S. Gov't Printing Office, Wash., DC, 1983.

Adams, Norman, and Rees, Jonathan. "Object-Oriented Programming in Scheme". *Proc. 1988 ACM Lisp and Funct. Progr. Conf.*, Snowbird, UT, 1988,277-288.

Agha, Gul. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Camb., MA, 1986.

Aho, A.V., *et al*. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

Alagic, Suad. *Object-Oriented Database Programming*. Springer-Verlag, New York, 1989.

ANSI-C. *Draft Proposed American National Standard Programming Language C*. ANSI, New York, 1988.

Atkinson, Malcolm P., and Buneman, O. Peter. "Types and Persistence in Database Programming Languages". ACM *Computing Surveys 19*,2 (June 1987),105-190.

Atkinson, M.P., Buneman, P., and Morrison, R. (eds.) *Data Types and Persistence*. Springer-Verlag, Berlin, 1988.

Atkinson, M.P., Bancilhon, F., DeWitt, D., Dittrick, K., Maier, D., and Zdonik, S. "The Object-Oriented Database System Manifesto". *Proc. First Deductive and Object-Oriented Database Conf.*, Kyoto, Japan, Dec. 1989.

Autodesk. *AutoLISP® Release 10 Programmer's Reference*. TD111-05.2, Autodesk, Inc., Sausalito, CA, 1988.

Backus, J. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs". *CACM 21*,8 (Aug. 1978),613-641.

Baker, Henry, and Hewitt, Carl. "The Incremental Garbage Collection of Processes". *Proc. ACM Symp. on AI and Prog. Langs., Sigplan Notices 12*,8 (Aug. 1977),55-59.

Baker, Henry. "List Processing in Real Time on a Serial Computer". *CACM 21*,4 (April 1978),280-294.

Baker, Henry. "A Source of Redundant Identifiers in Pascal Programs". *ACM Sigplan Not. 15*,2 (Feb. 1980),14-16.

Baker90a: Baker, Henry. "Unify and Conquer (Garbage, Updating, Aliasing...) in Functional Languages". *Proc. 1990 ACM Conf. on Lisp and Funct. Prog.*, June 1990.

Baker90b: Baker, Henry. "Efficient Implementation of Bit-vector Operations in Common Lisp". ACM *LISP Pointers 3,*2-3-4 (April-June 1990),8-22.

Baker90d: Baker, Henry. "The Nimble Type Inferencer for Common Lisp-84". Tech. Rept., Nimble Comp., 1990.

Baker91a: Baker, Henry. "Structured Programming with Limited Private Types in Ada: Nesting is for the Soaring Eagles". *ACM Ada Letters XI*,5 (July/Aug. 1991),79-90.

Baker91b: Baker, Henry. "Shallow Binding Makes Functional Arrays Fast". *ACM Sigplan Not. 26*,8 (Aug. 1991),145-147.

Baker91c: Baker, Henry. "Object-Oriented Programming in Ada83—Genericity Rehabilitated". *ACM Ada Letters XI*,9 (Nov./Dec. 1991),116-127

Baker92a: Baker, Henry. "CONS Should not CONS its Arguments, or A Lazy Alloc is a Smart Alloc". ACM *Sigplan Notices 27*,3 (March 1992),24-34.

Baker92b: Baker, Henry. "The Buried and Dead Binding Problems of Lisp 1.5: Sources of Incomparability in Garbage Collector Measurements". ACM *Lisp Pointers V*,2 (Apr-June 1992), 11-19.

Bawden, A. Pure Scheme emulation of cells from network mail circa 1988. Obtained from M. Felleisen.

Bird, R.S. "Tabulation Techniques for Recursive Programs". *ACM Comp. Surveys 12*,4 (Dec. 1980),403-417.

Bloom, Toby. "Immutable Groupings". CLU Design Note 61, MIT LCS, Aug. 16, 1976.

Bloss, A. "Update Analysis and the Efficient Implementation of Functional Aggregates". *Proc. 4th ACM/IFIP Conf. Funct. Progr. & Comp. Arch.*, London, Sept. 1989,26-38.

Burton, F. Warren. "A Note on Higher-Order Functions versus Logical Variables". *Info. Proc. Let. 31* (1989),91-95.

Cardelli, Luca, and Wegner, Peter. "On Understanding Types, Data Abstraction, and Polymorphism". ACM *Computing Surveys 17*,4 (Dec. 1985),471-522.

Carriero, N., and Gelernter, D. "Linda in Context". *CACM 32*,4 (1989),444-459.

Cohen, J.M., and Cohen, M.J. *The Penguin Dictionary of Quotations*. Penguin Books, Middlesex, England, 1960.

Cointe, Pierre. "Metaclasses are First Class: the ObjVlisp Model". *Proc. OOPSLA '87, Sigplan Notices 22*,12 (Dec. 1987),156-167.

Digitalk, Inc. *Smalltalk/V 286 Tutorial and Programming Handbook*. Digitalk, Inc., Los Angeles, CA, 1988.

DoD. *"STEELMAN": Department of Defense Requirements for High Order Computer Programming Languages,* June 1978.

Ershov, A.P. "On Programming of Arithmetic Operations". *Doklady, AN USSR 118*,3 (1958),427-430, transl. Friedman, M.D., *CACM 1*,8 (Aug. 1958),3-6.

Gabriel, R.P., and McCarthy, J. "Queue-Based Multi-Processing Lisp". *Proc. 1984 ACM Symp. on Lisp and Funct. Prog.*, (Aug. 1984),25-44.

Gabriel, R.P. "The Why of Y". *ACM Lisp Pointers 2*,2 (Oct.-Dec. 1988),15-25.

Gifford, David K., and Lucassen, John M. "Integrating Functional and Imperative Programming". *Proc. 1986 ACM Conf. on Lisp and Funct. Progr.*, Aug. 1986,28-38.

Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation*. McGraw-Hill, New York, 1983.

Goto, E. "Monocopy and Associative Algorithms in an Extended Lisp". Univ. of Tokyo, May 1974.

Goto, E., and Kanada, Y. "Recursive Hashed Data Structures with Applications to Polynomial Manipulations". *SYMSAC 76*.

Graube, Nicolas. "Reflexive Architecture: From ObjVLisp to CLOS". *Proc. ECOOP '88*, Springer-Verlag, Berlin, 1988,110-127.

Gunn, H.I.E., and Morrison, R. "On the Implementation of Constants". *Info. Proc. Let. 9*,1 (1979),1-4.

Guttag, John. "Abstract Data Types and the Development of Data Structures". *CACM 20*,6 (June 1977),396-404.

Halstead, R. "Implementation of MultiLisp: Lisp on a multiprocessor". *Proc. 1984 ACM Conf. on Lisp and Funct. Prog.*, (Aug. 1984),25-43.

Halstead, R. "MultiLisp: A language for concurrent symbolic processing". ACM *TOPLAS 7*,4 (Oct. 1985),501-538.

Harper, R., *et al.* "Standard ML". Tech. Rept. ECS-LFCS-86-2, Comp. Sci. Dept., Edinburgh, UK, March, 1986.

Harrison, Luddy, and Padua, David A. "PARCEL: Project for the Automatic Restructuring and Concurrent Evaluation of Lisp". *Proc. 1988 Conf. on Supercomputing*, St. Malo, France, 1988,527-538.

Herlihy, M., and Liskov, B. "A Value Transmission Method for Abstract Data Types". *ACM TOPLAS 4*,4 (Oct. 1982),527-551.

Hewitt, Carl, and Atkinson, Russell. "Synchronization in Actor Systems". *Proc. POPL 4* (Jan. 1977),267-280.

Hewitt, Carl E., and Baker, Henry. "Actors and Continuous Functionals". *Proc. IFIP Working Conf. on Formal Descr. of Progr. Concepts*, Aug. 1977, in Neuhold, Erich ed., *Formal Description of Programming Concepts*. North-Holland, Amsterdam, 1978,367-390.

Hilden, J. "Elimination of Recursive Calls using a Small Table of 'Randomly' Selected Function Values". *BIT 16* (1978),60-73.

Hillis, W. Daniel. *The Connection Machine*. The MIT Press, Cambridge, MA, 1985.

Hudak, P., and Bloss, A. "The aggregate update problem in functional programming systems". *Proc. 12'th ACM POPL*, Jan. 1985.

Hudak, P. "A Semantic Model of Reference Counting and its Abstraction". *Proc. 1986 ACM Lisp and Funct. Progr. Conf.*, Camb. MA,351-363.

Hudak, Paul. "Conception, Evolution, and Application of Functional Programming Languages". ACM *Computing Surveys 21*,3 (Sept. 1989),359-411.

Hughes, G.E., and Cresswell, M.J. *An Introduction to Modal Logic*. Methuen and Co., London, 1968.

Hughes, John. "Lazy Memo-functions". *Proc. Funct. Progr. & Computer Arch.*, Nancy, France, 1985,129-146.

Kahan, W. "Branch Cuts for Complex Elementary Functions, or Much Ado about Nothing's Sign Bit". Proc. Jt. IMA/SIAM Conf. on The State of the Art in Numerical Analysis, U. Birmingham, April 1986, Iserles, A., and Powell, M.J.D., Eds., Clarendon Press, Oxford, 1987.

Kale, L.V. "The Chare Kernel Parallel Programming System". *Int'l. Conf. on Parallel Programming*, Aug. 1990.

Katz, Morris J. *ParaTran: A Transparent, Transaction Based Runtime Mechanism for Parallel Execution of Scheme*. M.S. Thesis, MIT, Camb., MA, June 1986.

Keller, R.M., and Lindstrom, G. "Toward Function-Based Distributed Database Systems". Tech. Rep. 82-100, Dept. of Computer Sci., U. Utah, Jan. 1982, 37p.

Kent, William. "A rigorous model of object reference, identity, and existence". *J. O.-O.Progr.*, (June 1991),28-36.

Khoshafian, Setrag N., and Copeland, George P. "Object Identity". *Proc. OOPSLA '86, Sigplan Notices 21*,11 (Nov. 1986),406-416.

Kim, Won, and Lochovsky, Frederick H., *eds*. *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley, Reading, MA, 1989.

King, Roger. "My Cat is Object-Oriented". in [Kim89],23-30.

Klimov, Andrei V. "Dynamic Specialization in Extended Functional Language with Monotone Objects". *Proc. ACM PEPM'91*, New Haven, CT, June 1991, 199-210.

Knight, Tom. "An Architecture for Mostly Functional Languages". *Proc. 1986 ACM Conf. on Lisp and Funct. Prog.*, (Aug. 1986),105-112.

Kranz, David, *et al*. "Orbit: An Optimizing Compiler for Scheme". *Proc. Sigplan '86 Symp. on Compiler Constr*. (June 1986),219-233.

Krasner, Glenn, *ed*. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA, 1983.

Lamb, D.A., and Hilfinger, P.N. "Simulation of Procedure Variables using Ada Tasks". *IEEE Trans. Soft. Eng. SE-9*,1 (Jan. 1983),13-15.

Lang, Kevin J., and Pearlmutter, Barak A. "Oaklisp: An Object-Oriented Scheme with First Class Types". *OOPSLA '86, Sigplan Notices 21*,11 (Nov. 1986),30-37.

Larus, James Richard. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. Ph.D. Thesis, UC Berkeley, also published as Rep. No. UCB/CSD/89/502, May, 1989.

Lieberman, H., and Hewitt, C. "A Real-Time Garbage Collector Based on the Lifetimes of Objects". *CACM 26*,6 (June 1983),419-429.

Lieberman, Henry. "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems". *Proc. ACM/Sigplan OOPSLA'86*,214-223.

Liskov, *et al*. "Abstraction Mechanisms in CLU". *CACM 20*,8 (Aug. 1977),564-576.

Lomet, David B. "Objects and Values: The Basis of a Storage Model for Procedural Languages". *IBM J. Res. & Dev. 20*,2 (March 1976),157-167.

Lucassen, John M. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. Ph.D. Thesis, also MIT/LCS/TR-408, MIT, 1987,153p.

Lyngbaek, Peter, and Vianu, Victor. "Mapping a Semantic Database Model to the Relational Model". *Proc. ACM SIGMOD 1987 Conf., Sigmod Record 16*,3 (Dec. 1987),132-142.

MacLennan, Bruce J. "Values and Objects in Programming Languages". *Sigplan Not. 17*,12 (Dec. 1982),70-79.

MacLennan, Bruce J. "A Simple Software Environment based on Objects and Relations". *Proc. ACM Sigplan Symp. on Lang. Issues in Progr. Envs., Sigplan Not. 20*,7 (July 1985),199-207.

Maier, David, *et al*. "Development of an Object-Oriented DBMS". *OOPSLA '86, Sigplan Notices 21*,11 (Nov. 1986),472-482.

Mason, Ian A. *The Semantics of Destructive Lisp*. Ctr. for the Study of Language and Info., Stanford, CA, 1986.
McCarthy, J. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". *CACM 3*,4 (1960),184-195.
McCarthy, J., *et al*. *LISP 1.5 Programmer's Manual*. MIT Press, Camb., MA, 1965.
Milner, Robin. "A Theory of Type Polymorphism in Programming". *JCSS 17* (1978),348-375.
Mitchell, J.C. and Plotkin, G.D. "Abstract Types Have Existential Type". ACM *TOPLAS 10*,3 (July 1988),470-502.
Moon, David A. *MacLisp Reference Manual, Rev. 0*. Proj. MAC, MIT, April 1974.
Moon, D.A. "The Common Lisp Object-Oriented Programming Language Standard". in Kim, W., and Lochovsky, F.H., *eds. Object-Oriented Concepts, Databases and Applications*. Addison-Wesley, Reading, MA, 1989,49-78.
Morris, J.H. *Lambda-Calculus Models of Programming Languages*. Ph.D. Thesis, MIT, 1968.
Moses, Joel. "The Function of FUNCTION in Lisp". Memo 199, MIT AI Lab., Camb., MA, June 1970.
Mostow, J., and Cohen, D. "Automating Program Speedup by Deciding What to Cache". *Proc. IJCAI-85*, L.A., CA, Aug. 1985, 165-172.
Novak, G.S., Jr. "Data Abstraction in GLISP". *Proc SIGPLAN'83, Sigplan Notices 18*,6 (June 1983),170-177.
Ohori, A., Buneman, P., and Breazu-Tannen, V. "Database Programming in Machiavelli—a Polymorphic Language with Static Type Inference". *Proc. 1989 Sigmod Conf.*, Portland, also *Sigmod Record 18*,2 (June 1989(,46-57.
Ohori, Atsushi. "Representing Object Identity in a Pure Functional Language". *Proc. 3rd Int'l. Conf. on Database Theory*, Paris, Dec. 1990.
Pacini, G., and Simi, M. "Testing Equality in Lisp-like Environments". *BIT 18* (1978),334-341.
Padget, Julian, and Nuyens, Greg. The EuLisp Definition, Version 0.6. Univ. of Bath, Bath, Eng., July, 1989.
Plotkin, G.D. "Call-by-name, call-by-value, and the lambda-calculus". *Theor. Comput. Sci. 1* (1975),125-159.
Pratt, V.R. "Process Logic". *ACM POPL 6*, (1979),93-100.
Pugh, William. "An Improved Replacement Strategy for Function Caching". *Proc. ACM Conf. on Lisp & Funct. Progr.*, Snowbird, UT, July, 1988,269-276.
Queinnec, Christian, and Cointe, Pierre. "An Open Ended Data Representation Model for Eu_Lisp". *Proc. 1988 ACM Lisp and Funct. Progr. Conf.*, Snowbird, UT, 1988,298-308.
Radin, George. "The Early History and Characteristics of PL/I". *ACM Sigplan History of Prog. Langs. Conf., Sigplan Not. 13*,8 (Aug. 1978),227-241.
Rees, J. and Clinger, W., *et al*. "Revised Report on the Algorithmic Language Scheme". *Sigplan Notices 21*,12 (Dec. 1986),37-79.
Robinson, J.A., and Sibert, E.E. "LOGLISP: Motivation, Design, and Implementation". In Clark, K.L., and Tärnlund (eds), *Logic Programming*, Academic Press, 1982,299-314.
Ruf, Erik, and Weise, Daniel. "Nondeterminism and Unification in LogScheme: Integrating Logic and Functional Programming". *Proc. 4'th ACM Funct. Prog. Langs. and Computer Arch.*, Sept. 1989,327-339.
Sandewall, Erik. "A Proposed Solution to the FUNARG Problem". 6.894 course notes, MIT AI Lab., 1974. This solution was used in the MIT Lisp Machine [Greenblatt].
Sandewall, Erik. "Ideas about Management of Lisp Data Bases". AI Memo 332, MIT AI Lab., May 1975; also *Proc. IJCAI 4* (1975),585-592.
Sandewall, Erik. "Programming in an Interactive Environment: the Lisp Experience". ACM *Computing Surveys 10*,1 (March 1978),35-71.
Schwartz, J.T. "Optimization of very high level languages—I. Value transmission and its corollaries". *J. Computer Lang. 1* (1975),161-194.
Schwartz, J.T. "Optimization of very high level languages—II. Deducing relationships of inclusion and membership". *J. Computer Lang. 1*,3 (1975),197-218.
Snyder, Alan. "Encapsulation and Inheritance in Object-Oriented Programming Languages". *Proc. ACM/Sigplan OOPSLA'86*,38-45.
Steele, Guy L., Jr. *Rabbit: A Compiler for SCHEME (A Study in Compiler Optimization)*. AI-TR-474, AI Lab., MIT, May 1978.
Steele, G.L., and Hillis, W.D. "Connection Machine Lisp: Fine-grained Parallel Symbolic Processing". *1986 ACM Conf. on Lisp and Funct. Prog.*, (Aug. 1986),279-297.
Steele, G.L. "Fast Arithmetic in Maclisp". *Proc. 1977 Macsyma User's Conf.*, NASA Sci. and Tech. Info. Off. (Wash., DC, July 1977),215-224. Also AI Memo 421, MIT AI Lab., Camb., MA.
Steele, G.L., and Sussman, G.J. "The Art of the Interpreter, or The Modularity Complex (Parts Zero, One, and Two)". MIT AI Memo 453, May 1978, 73p.
Steele, G.L. *Common Lisp: the Language*. Digital Press, Burlington, MA, 1984.
Steele, G.L. *Common Lisp: the Language—Second Edition*. Digital Press, Bedford, MA, 1990.
Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

Swanson, Mark R., *et al*. "An Implementation of Portable Standard Lisp on the BBN Butterfly". *Proc. 1988 ACM Lisp and Funct. Progr. Conf.*, Snowbird, UT, July 1988,132-141.
Swinehart, D., *et al*. "A Structural View of the Cedar Programming Environment". ACM *TOPLAS 8*,4 (Oct. 1986),419-490.
Teitelman, Warren. *Interlisp Reference Manual*. Xerox Palo Alto Research Center, 1974.
Teitelman, W., and Masinter, L. "The Interlisp programming environment". *Computer 14*,4 (Apr. 1981),25-34.
Tinker, Pete, and Katz, Morry. "Parallel Execution of Sequential Scheme with ParaTran". *Proc. 1988 ACM Conf. on Lisp and Funct. Prog.*, (July 1988),28-39.
Turner, D.A. "A new implementation technique for applicative languages". *SW—Pract.&Exper. 9* (1979),31-49.
Ullman, Jeffrey D. *Principles of Database Systems*. Computer Science Press, Potomac, MD, 1980.
Ungar, David, and Smith, Randall B. "Self: The Power of Simplicity". *Proc. '87 OOPSLA, Sigplan Notices 22*,12 (Dec. 1987),227-242.
van Wijngaarden, A., *et al*. "Revised Report on the Algorithmic Language Algol 68". *ACM Sigplan Not. 12*,5 (May 1977),1-70.
Verity, J.W., and Schwartz, E.I. "Software Made Simple—Will Object-Oriented Programming Transform the Computer Industry?". *Business Week* cover story, Sept. 30, 1991,92-100.
Vianu, Victor. "A Dynamic Framework for Object Projection Views". ACM *TODS 13*,1 (March 1988),1-22.
Warren, D.H.D., Pereira, L.M., and Pereira, F. "Prolog—the language and its implementation compared with Lisp". *Proc. Symp. on A.I. and Prog. Langs., Sigplan Not. 12*,8 (Aug. 1977),109-115.
Wegbreit, Ben. "The Treatment of Data Types in EL1". *CACM 17*,5 (May 1974),251-264.
Welsh, J., *et al*. "Ambiguities and insecurities in Pascal". *SW—Prac. & Exper. 7*,6 (1977),685-696.
Wiebe, Douglas. "A Distributed Repository for Immutable Persistent Objects". *Proc. OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),453-465.
Wise, David. "Matrix algebra and applicative programming". In Kahn, G., *ed.*, *Funct. Progr. Langs. and Computer Arch., Lect. Notes in Comp. Sci 274,* Springer-Verlag, 1987,134-153.
Wulf, W. A., *et al.* "BLISS: A Language for Systems Programming". *CACM 14*,12 (Dec. 1971),780-.
Young, J.W.A., *ed*. *Monographs on Topics of Modern Mathematics Relevant to the Elementary Field*. Longmans, Green & Co., 1911. Reprinted by Dover Publications, 1955.

